UiO : Department of Informatics
University of Oslo

# Panorama Video Tiling: Efficient Processing and Encoding of Tiles

Hoang Bao Ngo

Master's Thesis Autumn 2015

# Panorama Video Tiling: Efficient Processing and Encoding of Tiles

Hoang Bao Ngo

3rd August 2015

# Abstract

High bitrate video has been exceedingly popular in recently, especially in soccer and gaming tournaments. A system which records soccer games and generate panorama videos can be found in the Bagadus system[41]. The panorama videos are generated from a stadium located in Tromso, Norway, and they can be viewed real-time by a virtual camera[53] which was implemented for users where they can interact the video stream. The virtual cameras does not only let the users move the camera around the panorama video, it also has the capability to pan, tilt and zoom into the video. But not everyone has a high bandwidth to receive a whole panorama video, and with the increase of potential consumers due its popularity there would be scaling challenges. Thus in this thesis, we introduce an encoder pipeline that is able to divide a generated cylindrical panorama video from the Bagadus System into tiles and each tile is encoded into different qualities. The overhead of tiling will be analysed extensively and different approaches to reduces storage consumption are also proposed.

# Contents

# List of Figures

# List of Tables

# Preface

I want to thank my supervisors Vamsi and Pål for giving me the chance to experience and complete my thesis, it has been a fun ride. In addition I want to thank Ragnar who also gave me alot of inputs which helped me start the project at the initial phase. When I encountered a problem using NVENC, he guided me through it hence I could finish the thesis. I would also like to thank Baard Winther, one of my colleague in the same room, for providing a simplified algorithm to identify a tile. Even though I found discovered a new one which is compatible with every tile formation, or at least those I've tried. I would also like to thank my colleagues whom are doing or are done with their thesis and wish everyone whether they are mentioned or not good luck and stay safe. Oh, and of course I want to thank my family, friends for sticking up with me through life.

Yeah, Karsten was it, thank you for proposing the use of Constant Rate Factor. I intended to implement a down and up scaling scheme, thus I saved a lot of time with your tip.

# Chapter 1

# Introduction

## 1.1 Background

The idea of tiling has been very popular in the last decade. It was been implemented in various scenarios, like windows 8 where they use tiles in start screen, or in applications such as SmartView[48] where you can stream several videos from different sources at the same time. The interest has been huge as a consequence of how fast technology has progressed. The current mainstream high definition standard is 1080p, but in the last few years Ultra High Definition(UHD) and 4K resolution are becoming morea nand more popular. Even one of the largest corporation in the technology industry, Toshiba, suggested that 4K Television will be mainstream by 2017[1]. There are also 8K resolution or Full Ultra High Definition(FUHD) which is being developed, though it will probably not be mainstream in the nearest future, but we can still notice how fast digital video are evolving. With high resolution videos, transferring the stream will require a lot of bandwidth and thus introduces huge scaling problems. Video compression technology has been the solution to handle the challenges of sending high bitrate streams to users. By reducing the bandwidth requirement it can scale to a large number of consumers, but there is a limit of how much data it can compress before visual artifacts will be shown. Thus leads us back to tiling. Tiling was another approach to further reduce the consumption of bandwidth without using to much compression, which could affect the visual acuity of the video.

## 1.2 Problem Definition / Statement

As a result of high spatial resolution video being more popular, there has been a trend to implement interactive virtual cameras where users can control it and move around a panorama video, thus making them be their own camera man. In the Bagadus system[41], such a virtual camera[53] was implemented for users where they can view real-time panorama videos, which are generated from a soccer stadium located in Norway Tromsø in real-time. The virtual cameras does not only let the users move the camera around the panorama video, it also has the capability to pan, tilt and zoom into the video. Not everyone has a high bandwidth to receive a whole panorama video, so a concept was proposed to handle the issue. It was possible to exploit the fact that a virtual camera usually has a limited vision when moving around, thus the solution was to tile a high bitrate video stream up into equivalent sizes of tiles. Based on the where the camera is focused, we will update the area where the camera is focused on with the high quality video tiles, and send lower quality tiles outside of its view. Thus we can lower the bandwidth requirement.

The problem we are trying to solve is to reduce the bandwidth requirement by producing tiles with different qualities and if possible be able to stream the encoded video file in real-time. Thus there are many other factors that comes into play, such as how do we tile a panorama video. Which approach or resolution of a tile is optimal. How much overhead can we expect from such a tiling system where a tile will be encoded into different qualities. Other factors that

must be considered is storage consumption, by having several video streams with the same content but in different qualities would require a lot of space. Can it be streamed in real-time?. Thus in this thesis we will explore the different video compression technologies available to find out which are the most optimal to use for encoding. Experiments with different methods to further reduce the bandwidth consumption while maintaining real-time encoding requirement if possible.

## 1.3 Limitations

Even though the goal of the thesis was to create a server which could receive a raw panorama video directly from a source and encode them into different qualities, we had to design the system to work locally to reduce the complexity of the implementation. Thus a limitation is we can only try to simulate a process where the server can receive streams from source. Another limitation is field testing, we are using FFmpeg to process video data, but the libraries are full of memory leaks thus it will crash eventually at some point in the future. We are testing hardware encoders such as NVENC, but there has been a restriction on the number of encodings session after a driver update from NVIDIA. Thus we had to downgrade many drivers to be able to use it for our cause. This could lead to unsatisfactory result as a consequence of issues or bugs that was supposed to be removed in newer updates.

## 1.4 Research Method

The first step we did was to define the requirements and specifications which the implementation of our program should be able to conform to. Then we will begin with researching relevant work within the field and make a design, and based on the design we will implement a prototype. The demo program will be tested extensively to find out if it is possible to optimize further. During this thesis we are expecting to implement a lot of prototypes. Each of them will be experimented with and based on the results, we will proceed to modify our design and attempt to improvise the prototype further. This procedure will be repeated until we get results that can satisfy the requirement and specification we made. Our approach in this master thesis is based on the Design methodology as described by the ACM Task Force in Computing as a discipline[10].

## 1.5 Main Contributions

The main contributions in this thesis has been the implementation of a tiling system where panorama video can be separated into smaller equivalent sizes and encoded into different qualities. Thus could potentially reduce the bandwidth consumption. We have also contributed a design where each part of the process is explained in detail for implementation of a working prototype.

1. A tiling algorithm
2. Analysis of different tiling approaches
3. Analysis of the overhead cost produced as a consequence of tiling
4. Analysis of the software encoder x264 used in combination with FFmpeg
5. Methods to reduce the latency in x264
6. Techniques to reduce the storage consumption thus bandwidth
7. Analysis of the hardware encoder NVENC

8. Techniques to optimize NVENC
9. Demonstrating how well the system can scale up by integrating hardware components

We also have a paper in proceedings for the 21st International Packet Video Workshop (PV 2015)[17].

## 1.6 Outline

In the first chapter we wrote about our motivation behind this thesis, found the problems that has to be solved with the limititions we have. Then we mentioned about our reasearch method and the main contributions in this thesis.

**In chapter 2 – Adaptive Encoding of Video Tiles** We will be discussing various related work in tiling which focuses in the server. We will also mention technologies which we used as a concept and adopted it to our design.

**Chapter 3 - Software Encoding: Design and Implementation** In chapter 3 we will introduce various technologies and methods to reduce the bandwidth requirement and how to implement a working software encoder

**Chapter 4 - Case Study: Software Encoding with libx264** In chapter 4 we will test out different tiling approaches and find the potential overhead caused by tiling and the reduction of bandwidth.

**Chapter 5 - Hardware Encoding: Design and Implementation** We will introduces some hardware encoders, how to use them, limitations they have and how to implement a working prototype

**Chapter 6 - Case Study: Hardware Encoding** We will be experimenting the hardware encoder and evaluate the results

**Chapter 7 - Case Study: Software Encoding combined with Hardware Encoding** In chapter 7e will introduce and comfirm the scalability of our implementation

**Chapter 8 - Conclusion** In chapter 7 we will evaluate the design and draw some final conclusions and mention some technologies and suggest new approaches to which could be used to further develop it.

# Chapter 2

# Adaptive Encoding of Video Tiles

## 2.1 Tiling Concept

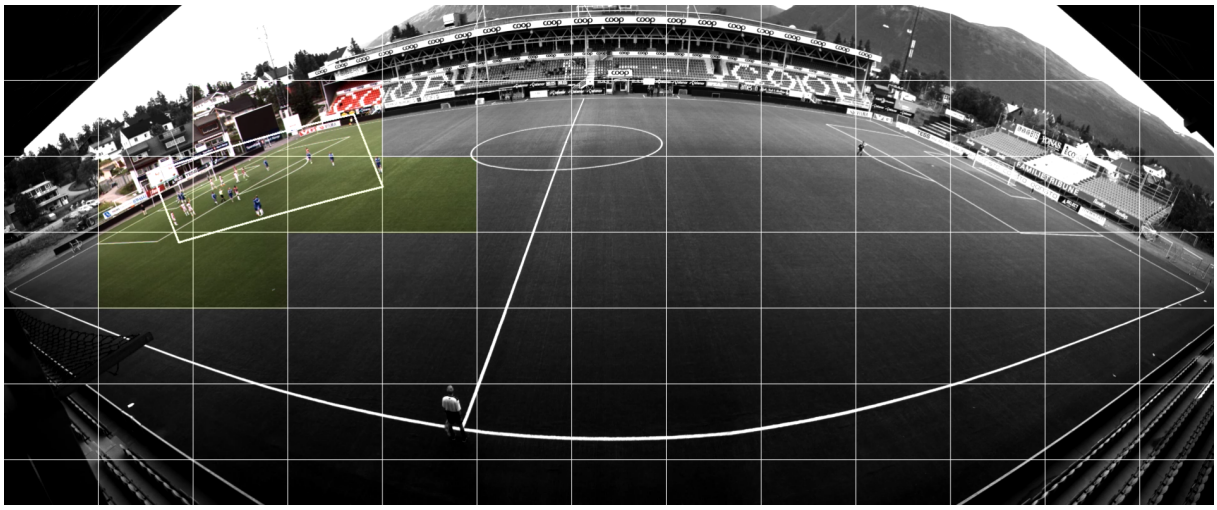


Figure 2.1: Illustration of how Interactive Virtual Camera works[49]

In figure 2.1, it shows us the concept of how an interactive virtual camera in the Bagadus[41] system works. The tiles which are in the Region of Interest are the coloured ones, they depict high quality tiles while the greyed out tiles are of low qualities. Using RoI to decide which quality of tiles to fetch in accordance to the camera are adapted to many implementations of virtual cameras to reduce the bandwidth requirement, to make it more scalable. The way interactive virtual camera shown in the aforementioned figure works is, only fetch the tiles overlapped with the RoI in high quality and all other tiles outside of RoI in lower quality. The outer tiles will be fetched in higher qualities subsequently depending of the bandwidth of the user. Thus one of our goal is to use tiling to reduce the overhead and potential bandwidth by exploring different approaches of compression technologies. There has also been researched extensively on how to reduce the bandwidth consumption from the server side which are mentioned in [2], [23] and [30].

## 2.2 Adaptive bitrate streaming

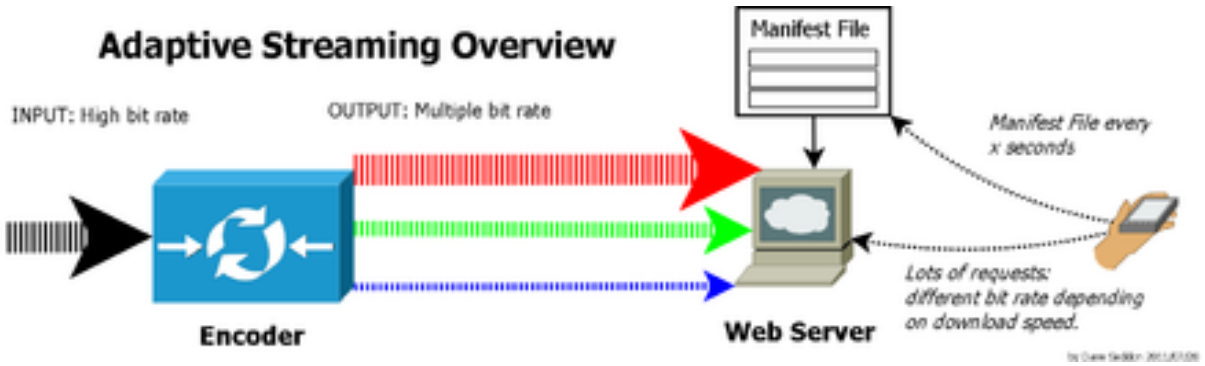



Figure 2.2: Overview of how adaptive streaming works [3]

Adaptive bitrate streaming [3] is a technique used for streaming of multimedia content over computer networks. Historically, video streaming technologies were built on streaming protocols such as Real-time Transport Protocol(RTP), Real-Time Streaming Protocol(RTSP) or RTP with RTSP, but nowadays most adaptive streaming approaches are based on Hypertext Transfer Protocol(HTTP). It is designed for transmission over large distributed HTTP networks in an effective and efficient way. The user or client in our case are devices that are able to playback video streams, for example a television, personal computer or mobile devices such as mobile phones or tablet computers. The way it works is by monitoring the user's bandwidth, CPU and memory capacity in real-time and then making corresponding adjustments to the video quality. The core process involves an encoder which can encode a single source video at multiple bit rates, and then segmenting each of the different bit rates into small parts. The duration of the segments typically varies between 2 and 10 seconds, but that depends on the implementation. A manifest file will be created and it will contain the information of all the video segments and their bit rates. At the beginning, when a client accesses the multimedia file, it will request the segments from with the lowest bit rates from the given manifest file. The transmission between the server and the client is monitored, if the user detects that the download speed exceeds the bit rate of the initial downloaded segment, then it will request the next higher tier of bit rates segments. If the download speed deteriorates, the client will then adjust to the situation and request a lower category of bit rates segments. This process will keep going until the current bit rate segments are closely matched with the available bandwidth at the client. This is how the adaptive bitrate streaming technology can accommodate a wide range of devices and network capacities, and therefore can provide the users the best video quality and viewing experience on both low bandwidth and high bandwidth. The only downside with this technology is it requires more processing in the server side caused by additional encoding, but in return it is very flexible and simplifies the work flow. Various adaptations of adaptive bitrate streaming has been implemented by Microsoft Corporation, Adobe Systems, Apple Incorporated and Moving Picture Expert Group(MPEG).

## 2.3 Summary

The idea of tiling originated from the use of interactive virtual cameras as mentioned above. By utilizing the same concept in adaptive bitrate streaming we could be able to stream real time video and provide quality of service. Some server side processing has been proposed in [2], [23] and [30]. But the way they adapt to user is to either downscale a tile, then up scale. Cropping was also discussed, but none of them had a solutions for adaptive streaming of tiles. Thus we decided to not go into detail of each of the papers.

# Chapter 3

# Software Encoding: Design and Implementation

## 3.1 Introduction

There are several libraries and background theories that we have to be aware of, before we can proceed to designing and implementing our solution of tiling and investigate the overhead cost and bandwidth. One of the challenges we have with delivering of a high resolution and high bitrate panorama video to a huge number of users is scaling. The required resources such as processing power, storage and bandwidth would be huge. Many researches have already proposed many different solutions such as tiling, downscaling and up-scaling or cropping to reduce the resource requirement. One of the services the Bagadus system[20] offer is an interactive moving virtual camera to each user. Since the virtual camera has a limited view of the panorama video, we can use tiling to reduce the quality of every tiles other than the region of interest(ROI) depending on the position of the virtual view. Thus we can reduce the bandwidth requirement. The problem we have is that we have to be able to dynamically change the quality of the tiles and none of the existing tiling approaches dealt with this particular issue. In this section we will further discuss about the design and theories that are required to understand the concept.

## 3.2 Video compression

One way to reduce the consumption of bandwidth and storage is video compression. Essentially compression of video is a process that involves applying an algorithm to a source video, which reduces and removes redundant data from it and creates a compressed file so it can be effectively sent and stored. However, if the compression of a video is too high, it may lose too much data thus presenting visible artifacts during playback. A compressed video can not be viewed before we reverse the process. To play video files which are compressed(encoded), we need to decompress(decode) the file by applying an inverse algorithm to it thus producing a video that shows virtually the same content as the original source. The trade-off is between the latency or time it take to reverse a process, applying more advanced compression algorithm on a file will result in longer time in decompression. The video codec format we will be using is H.264, since it is supposedly exceptionally efficient in delivering good video quality at a lower bit rates than most other video compression technologies(such as the previous standards H.263, MPEG-4 Part 2 or MPEG-2)

## 3.3 H.264 / MPEG-4 AVC

H.264, also known as MPEG-4 Part 10, Advanced Video Coding(MPEG-4 AVC) is a video compression technology that was developed by ITU Telecommunication Standardization

Sector(ITU-T) Video Coding Experts Group VCEG)[29] as H.264 and International Organization for Standardization/International Electrotechnical Commission Moving Picture Experts Group(ISO/IEC MPEG-4)[28] as MPEG-4 AVC. The terms H.264 and MPEG-4 AVC are often interchangeable since the ITU-T H.264 standard and ISO/IEC MPEG-4 standard are maintained together thus they have identical content.

## 3.4 Choosing a software encoder

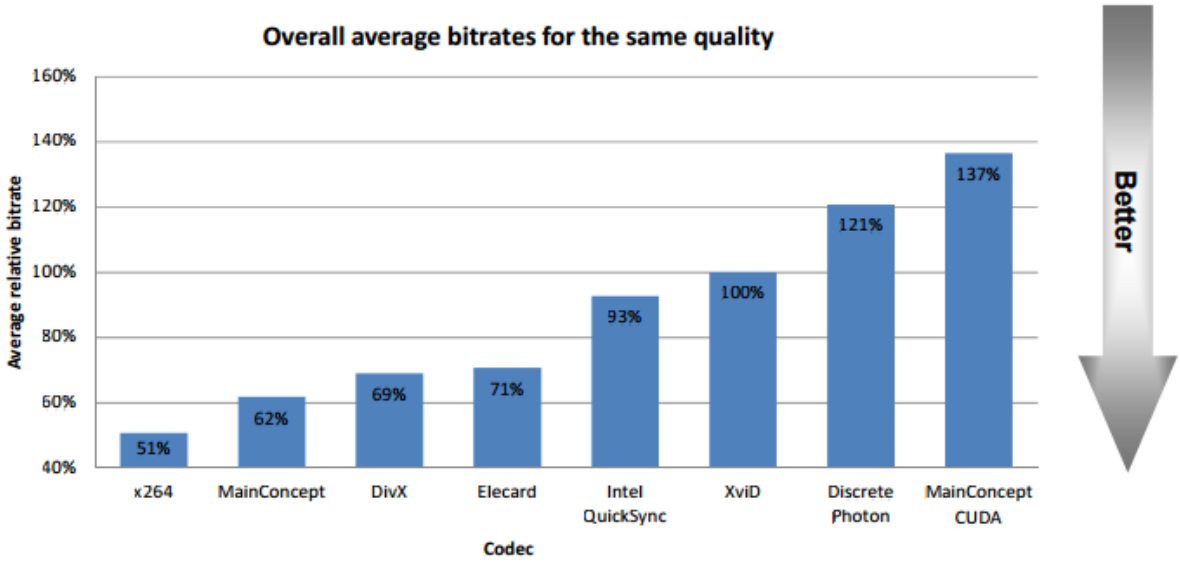



Figure 3.1: Average bit ratio for a fixed quality for all categories and all presets [19]

There are many AVC software implementations which has the ability to encode a video stream into H.264 format, some of are mentioned in figure 3.1. The qualities are measured with Y-SSIM, which is a variation of the standard SSIM, and according to the results x264 is one of the best codecs by encoding quality. Certainly the results does not necessarily correlate with subjective video quality, since the results can be adjusted by modifying the encoder settings hence giving higher metric scores. They mentioned in [19] that they used a special encoding option("tune-SSIM) in x264 which could explain the quality difference. So we had to do some initial tests with x264 before deciding whether we should experiments with other encoders. X264 was very user friendly, with its predefined presets, and it also had many encoding settings that can be easily adjusted. As such we decided it was the most suitable software encoder to use in our design and implementation.

### 3.4.1 x264, a H.264 encoder

x264 is an open source implementation which encodes video streams into H.264 file formats. The one we are using is called libx264 which is a "wrapper" around x264. FFmpeg does not have a native encoder for H.264 format thus it uses x264 as an external library for encoding. For it to work, FFmpeg implemented libx264 which converts FFmpeg generic data structures to a format x264 can accepts and process them. Thus when we are referring to using FFmeg or libx264, it means we are actually using an x264 encoder to process a file.

[0]Structural Similarity index is a method for measuring the similarity between two images[44], Y-SSIM only measure the Y channel(will be further explained later)

### 3.4.2 Constant rate factor

Constant rate factor or CRF, is a rate control scheme which are implemented in x264. We can set a value from 0-51 where 0 is lossless, 23 better and 51 worst. The lower number we pick the better compression will we get, but it will also require more processing time. The concept of CRF exploits how the human eye perceives motions. By deducing if a frame has a lot of motions in it, CRF will apply a higher Quantization Parameter(QP) on it. A QP decides how information is tossed by the encoder, the higher the QP value is the more informations is lost. However in still images CRF will apply a lower QP for better visual appearance. Thus we can reduce the data rate. In terms of visual perception, we would think it is constant due to the fact that our eyes would be distracted by everything happening, thus would not have enough time to notice the heavier compression. Constant Rate Factor is the 1 pass rate control which will be used in this design. Since we are tiling a panorama video into separate file, we could either crop or downscale a video stream to get a lower bitrate. However with CRF, we can apply it directly to a tile to get a lower bitrate while have better visual, even though in it is not if a machine evaluates it. We can see in figure 3.2 the loss of sharpness in the downscaled video.

Figure 3.2: Using CRF left 158 kbps, downscaled rigth 157kbps[17]

## 3.5 Techniques to reduce the storage space and bandwidth requirement

### 3.5.1 Frame Types

There are several types of frames, and these are intra coded pictures(I-frames), predictive coded pictures(P-frames) and bidirectionally predicted coded pictures(B-Frames). There is another one called D-pictures, but these are not commonly used so we are not going to explain about it.

I-frame are frames coded by themselves. It means they contain the full image, which can be decompressed without the need for additional data from any other frames. The disadvantage is they do not have good compression because the information they keep is important. However, the benefit of using these picture types is that the video stream is more editable. That means we can access the video at various points in the stream randomly, for example searching a specific scene or fast forwarding. The effect of not having an I-frame in a scene change is that the first few scenes will be visually blurry, but it will gradually get clearer after a short while. Another type of pictures is the P-pictures. They are forward predicted, which means they need information from the previous I-frame or P-frame to reconstruct the image. They do not require much space as I-frames, since they do not contain a full image. The information they hold is just a part of an image, more specifically they only contain the differences of motion relative to the previous decoded frame, so the unchanging background pixels are not stored. The last type of pictures are the B-frames. They are frames that requires information from both previous and next I- or P-frames for decompression. The information a B-frame contains is also the differences of motion between frames, but unlike P-frames the content of a current B-frame can be reconstructed by references from both the preceding frame and the following frame. Thus it saves even more space than both I-frames and P-frames.

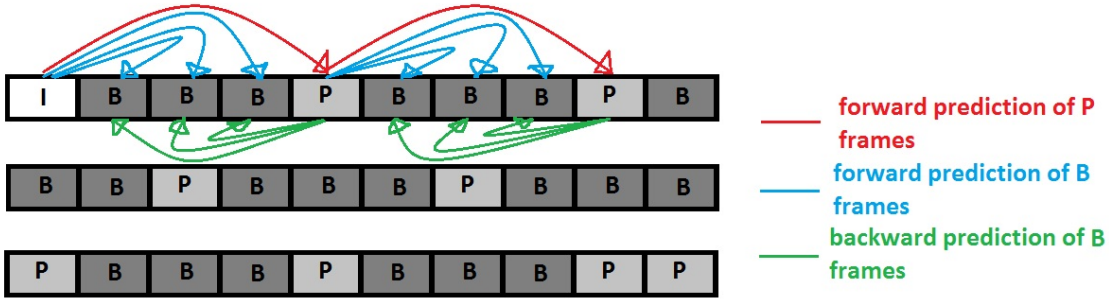



Figure 3.3: Overview of how the different picture type interacts with each other. This is the Group of Pictures structure of a 1 second video segment .

### 3.5.2 Group of Pictures

A Group of Pictures, also known as GOP, is a specific sequence of frames in a video stream. Each video stream consist of 1 or more Group of Pictures. We have mentioned in 3.5.1 that there are 3 types of picture frames; I-frames, B-frames and P-frames. GOP are determined by the pattern of these frame types, the size and whether it is open or closed. The sequence of structure a GOP is important since the sizes of each frame types are different. As we know from 3.5.1, the picture type that requires the most space are the I-frames, then the B-frames and lastly the P-frames. In general, the sequence of a GOP has many variations, but it usually starts with a I-frame. It is possible to have a B-frame at the beginning in an open GOP. Since it is allowed to have references from one GOP to an I-frame or P-frame in an adjacent GOP. The limitation is a B-frame can't be at the beginning in the first GOP or at the end in the last GOP(or in a GOP, since a GOP cannot end with a B-frame), since a B-frame needs information from both sides to decompress. A closed GOP will always begin with an I-frame and ends with a P-frame, and unlike an open GOP, it does not rely on frames outside of the GOP for decompression. Our goal is to reduce the bandwidth requirement, so the most important factor for us when choosing the GOP structure is to reduce the size. Overall size are generally determined by distance between each P-frames. If the P-frames are farther apart from each other then there would be more B-frames, thus resulting greater compression. The space between the frames can be determined by the difference of motion between a P-frame to the previous P- or I-frame. If there are minimal changes between the 2 frames then they can be farther apart from each other, if the difference is big the distance should be smaller between them. Another way to reduce the bandwidth is by setting the GOP size, it specifies number of frames in a GOP. By increasing the length of GOP, there would be less I-frames in a video stream. Since I-frames are larger in size than both B- and P-frames, we can reduce the consumption of storage space. There are any ways to set up the GOP, and the trade-off is complexity vs storage. For example, with only I-frames in a video stream with 30 frames(thus 30 GOP) the decompression would be simple, because each frame is not dependant of another frame. The downside is the required storage space would significantly increase . If we compress the video more by having greater GOP sizes, then the decompression would take more processing time since there are more B-frames and P-frames. Illustration of our GOP in figure 3.3

### 3.5.3 Segmentation and GOP size

| 2X2 | 90 GOP | 30 GOP |
|---|---|---|
| out21 | 5.3M | 7.4M |
| out24 | 4.0M | 5.5M |
| out30 | 1.6M | 2.5M |
| out36 | 696K | 1.3M |
| out48 | 222K | 428K |

Table 3.1: Table shows the difference of required storage space between using GOP 90 and GOP 30. The first column describe how many tiles we have and the numbering tells us which QP we encoded the tiles with.

We are trying to reduce the bandwidth to improve the scalability, thus the size of each video stream produced by the tiling system plays an important role. The segments in our system has a duration of 3 seconds and GOP size we used are 30 and 90. We tested on both lengths to give us an indication of the difference between having 3 GOP versus 1 GOP, and find out how much space we can potentially save. By increasing the GOP size we did indeed reduce the size of files significantly based on the results in table 3.1. The disadvantage, like we also mentioned in 3.5.2, is the trade-off between saving space and the complexity in decompression. The duration of the segments plays another important role, we can optimize the bandwidth and storage space consumption a lot by increasing both the duration of the segments and also increase the size of GOP. The overall quality adaptation latency will also increase proportionally with the duration of the segment though.

### 3.5.4 Ffmpeg

Fast Forward Motion Picture Expert Group(Ffmpeg) is a free software and it is a multimedia framework. It contains libraries and programs that enables us to handle every existing multimedia file ever created. We are only using libraries that are required to encode or transcode and decode video files. In our program we are converting a existing digital format to another digital format with different bit rate, thus based on the definition from this site [51], it is transcoding. We decided to simulate the process of encoding video stream directly from source in this kind of manner, since it is easier to work with digital files. Still, we want to be able to encode video files straight from source in the future, but unfortunately not in this thesis. We will proceed to mention some of the important structures we used from the ffmpeg library for reading a file, demultiplexing(demuxing), decoding, encoding and multiplexing. Functions from the library will be explained in relevant places.

## 3.6 Image Formats

### 3.6.1 YUV

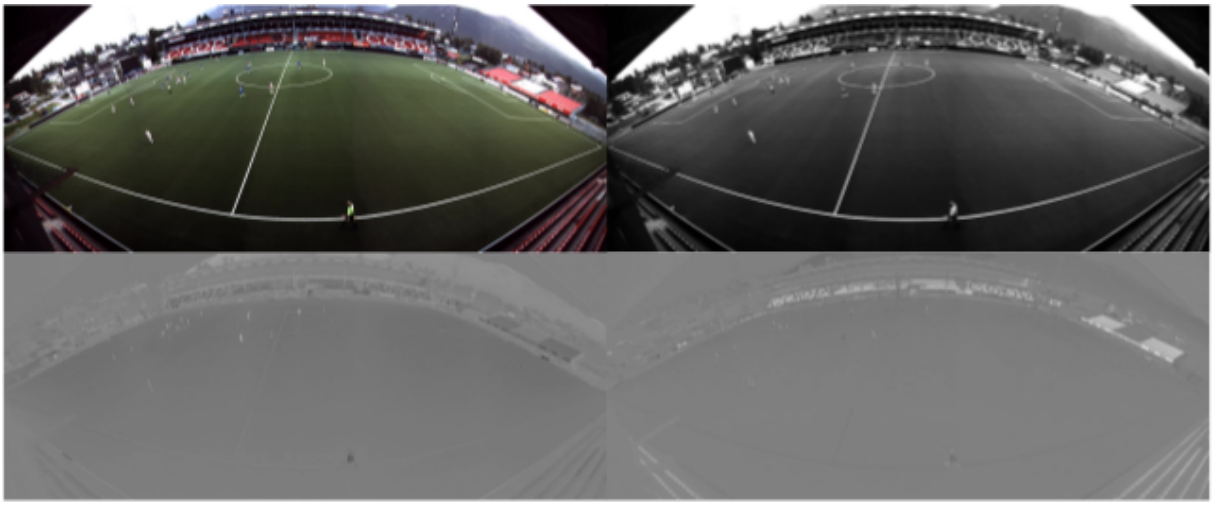
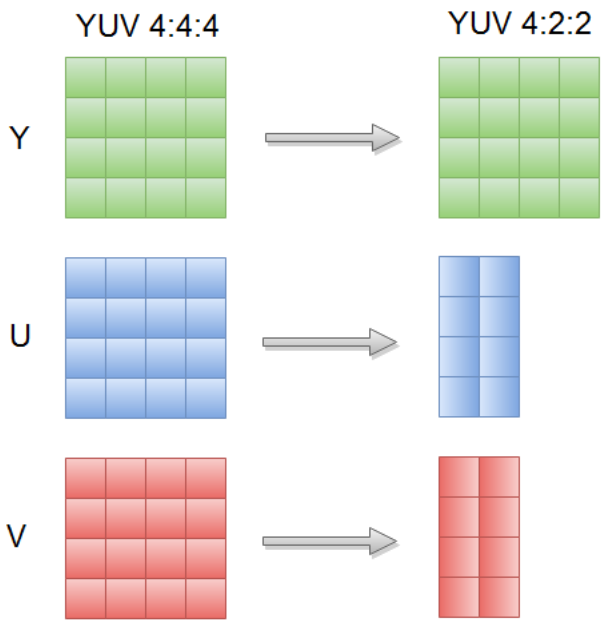


Figure 3.4: Illustration of YUV combined and each channel in YUV separated in greyscale

YUV is used to represent colors in videos and images. The Y, U and V component contains the luminance channel, and two chroma channels retrospectively. Humans are more sensitive to differences in luminance, which is the brightness, and less perceptive to changes to the chroma components. If we observe the figure 3.4, we can we the Y channel in the top right row, U in the bottom left row and right the V channel. They are represented in greyscale so it is easier for us to see any variance between the channels. From the picture we can observe how much detail the luminance channel contains compared to the chroma components, also the U and V channels when compared to each other does not have that much difference. Thus it is possible to downsample them(hence reducing the size) without losing any noticeable detail. Even if there are visual artifacts, we would not be able to observe it. As such YUV color space are usually used as a part of a video pipeline to produce video streams with a lower bitrate while retaining the visual quality. One such pipeline is found in the Bagadus systemwhere it generates panorama video streams from a soccer stadium. The videos can either be streamed in real-time or stored as raw H.264 files in pixel format YUV 4:2:2.

**YUV 4:2:2**




YUV 4:2:2 is a pixel format where the U and V channel are sampled half the rate of Y in the horizontal dimension. So if we use YUV 4:4:4 as a basis then YUV 4:2:2 can be seen in figure **??**. Thus it seems like we can reduce the bandwidth by 1/3 of the original size.

## 3.7 Image Stride

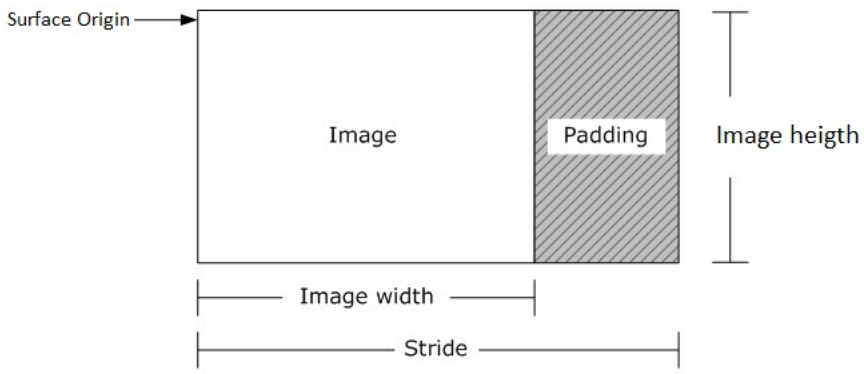



Figure 3.5: Image stride from [33] with a little change

A video stream can be seen as a sequence of images, each representing a single frame in the video. The frame can be represented as rows of pixels, and each row of pixels might contain extra padding. It is possible to use padding for scaling, such as reducing the image by the amounts of padding or increasing the image by filling the padded area with pixel informations. Padding affect on how the image is stored in memory, but it does not alter the displayed picture. The stride in an image is the width of the displayed image and the amount of padding the image has. The stride are usually larger than the image width since it might contain paddings. That is why we have to take the image stride into account whenever we process a video. The Surface Origin that was mentioned in figure 3.5 refers to the first byte of the image.

## 3.8 Initialization pipeline

### 3.8.1 Initializing the library and ensure thread safety in ffmpeg

Before we can use FFmpeg for encoding,decoding, muxing or demuxing of a media file, we have to initialize the corresponding format and codec of the file. There are two ways to do this, the first one is to register only certain individual file formats[1] and codecs[2](h264 and libx264) then manually initialize the parser and bitstream filters(which will be used in decoding and encoding), the second approach is to register all available file formats[3] and codecs[4] with the library. There is no benefit in choosing the first approach over the second, thus using the latter is preferred.
Thread safety was not considered until we began to encode multiple video stream in parallel. Some of the output files we got had visual artifacts, others did not. The cause is most likely the use multi-threading. Threads are commonly not referenced to one another and they usually share some data between them that needs to be frequently updated. In a non-threading environment, each instructions are executed in a proper sequence, but in parallel the order of instructions may be interleaved[5]. Thus whenever the executions are out of order we will get incorrect data. We found out the issues originated from several functions in FFmpeg library[13], so in order to encode several streams concurrently, we need to synchronize access on the functions by applying a locking mechanism. One of the solutions we found and tested was the use of av_lockmgr_register to register a mutex handler[6] in FFmpeg, which is then used whenever we call functions from the library. The other solutions that should also work is to build FFmpeg with –enable-pthreads, it will automatically synchronize function calls by using its own lock manager implementation.

### 3.8.2 Opening and decoding a Panorama Video - Demultiplexing and decoding module

As we mentioned before in , our encoder prototype does not have a feature for retrieving frames directly from the Bagadus system, thus we have to work with panorama videos that has already been encoded and stored locally as H.264. This part of the pipeline is responsible opening and reading a file, then extracting the contents and store it as raw YUV frames in memory. We decided to store the contents of a video file in memory to simulate an enviroment where our encoder is integrated in the Bagadus system, thus frames can be fetched straight from source without the need for reading,demuxing then decoding a file.

#### 1. Prototype - Sequential Encoding of Panorama Video

During development, we implemented a prototype for testing the FFmpeg library and to give us an estimation of the transcoding time. It could read a H.264 file stored on disk, then transcoded it to the same format and codec. It worked well and the transcoding time was under real-time requirement of 3 seconds when we had only 1 quality. We later added a new feature which transcoded an input file to multiple output files with different qualities. The time it took to transcode increased exponentially which was in our prediction, but we wanted to reduce the time so we explored various parts of the program to see if there are any areas which we could optimize. The first region which could affect the time measurement was file reading and

[1] use av_register_input_format(AVInputFormat *format) for input and av_register_output_format(AVOutputFormat *format) for output

[2] avcodec_register(AVCodec *codec)

[3] av_register_all(void) registers all format available in the library for muxing and demuxing, it will also execute avcodec_register_all(void)

[4] avcodec_register_all(void) registers all codec available in the library for encoding and decoding, it will also initialize the parser and bitstream filters at configuration time

[5] Race Condition[46]

[6] The locking implementation was found from [11] and it was also proposed by Vamsii

writing operations. This should not be a bottleneck since we have a Solid State Drive(SSD) as our storage medium, which makes these operations exceedingly fast. But we have to keep in mind that when reading the media file, it also processes the data read by decoding it thus using CPU-resources. This issue becomes more apparent when we added tiling and tried to encode several files in parallel which will be further discussed in section 3.8.4 and mentioned in prototype 2. The general flow of the first prototype is as follows:

1. Open input file
2. For as long as there are still tasks left
   (i) Create output file
   (ii) Open encoder for the newly created file
   (iii) For as long as there are still frames left in the input file
      (a) Fetch a frame from input file
      (b) Decode the frame and store the raw YUV data in memory
      (c) Encode the raw YUV data and write it to output file
   (iv) Close output file and encoder
   (v) Reposition the stream pointer to the beginning of the input file
3. Close input file

**Module Implementation - Initializing a file for reading**

Like we discussed in 3.8.2, the primary goal this module has is demuxing and decoding of Panorama videos. As the name implies, the procedure consist of 2 parts and the first one is discussed here. We are utilizing ffmpeg for the implementation. The libraries we used are libavformat(lavf) which contains demuxers and muxers for different audio and video container formats, and libavcodec which consist of encoders and decoders that can accommodate the majority of audio and video files. Some of the structures we will be using are:

1. AVFormatContext - A structure that stores all information of an opened file, all IO operations of files goes through here
2. AVStream - A structure that has information of a stream
3. AVCodecContext - A structure which contains codec information from a stream
4. AVCodec - A structure that contains information of a specified codec and function pointer for encoding and decoding

First we have to open the file which is stored as h264 before we can process the data. With lavf, we can open media files by passing the filename to function avformat_open_input(). It opens the specified file(automatically detects the input format if not provided), read the header and then store the extracted information in an AVFormatContext. Usually the next step is to find how many streams there are in the media file and store it in AVStream, but we know from , h264 files are a file extension which denote raw H.264(AVC[7]), hence there is only one stream. After we have extracted the codec information from the stream and stored it in AVCodecContext, we used avcodec_find_decoder() to find the corresponding decoder for the stream. Then we have to initialize it by using avcodec_open2() before using the decoder.

[7] Advanced Video Coding, which is a digital video-compression format

**Module Implementation - Reading from a file**

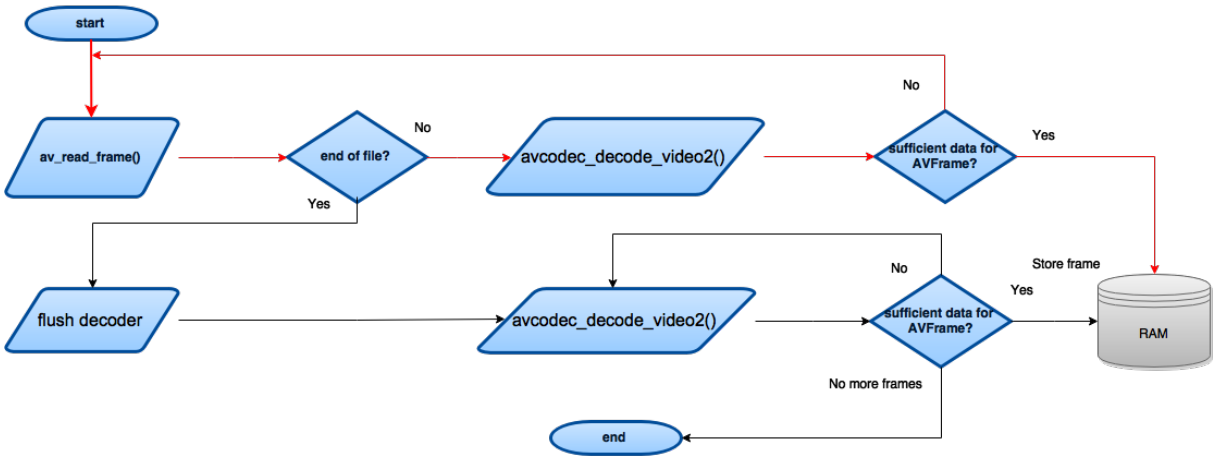



Figure 3.6: Overview of reading, decoding and storing the raw data

The illustration in 3.6 shows us the second part of the module implementation, which is to read data from an opened file, decode it then store it in memory. Some structures worth mentioning:

1. AVPacket A structure which contains encoded data and data which is used to identify the stream it originated from

2. AVFrame A structure which contains the decoded video data

We can read the entire data stream from an opened file by repeatedly calling av_read_frame(). It will return an AVPacket with references to where the encoded data is allocated and stored each time it is called. We can then decode the compressed data with avcodec_decode_video2() which will convert the AVPacket to an AVFrame. Usually it will take several AVPackets to have sufficient data to fill an AVFrame, especially if is the first frame(which is most likely an I-Frame), thus we will have to call the decode function several times while providing it with new data. We will be notified when an AVFrame has been decoded and proceed to store it in memory. We discovered that when end of file has been reached, there was still some frames left in avcodec_decode_video2(). These can be fetched by flushing the encoder which is repeatedly calling on the decoder until there is none left. The delay is a normal behaviour from ffmpeg and there are two reasons for it. The first one is for multi-threaded efficiency. During decoding one thread will queue frames to the decoder and other threads will fetch the data from the queue and decode them, thus even if file has reached the end there could still be frames that are in the middle of decoding. This situation is more likely to happen in the case of B-frames where the decode order may not be the same as the display order.

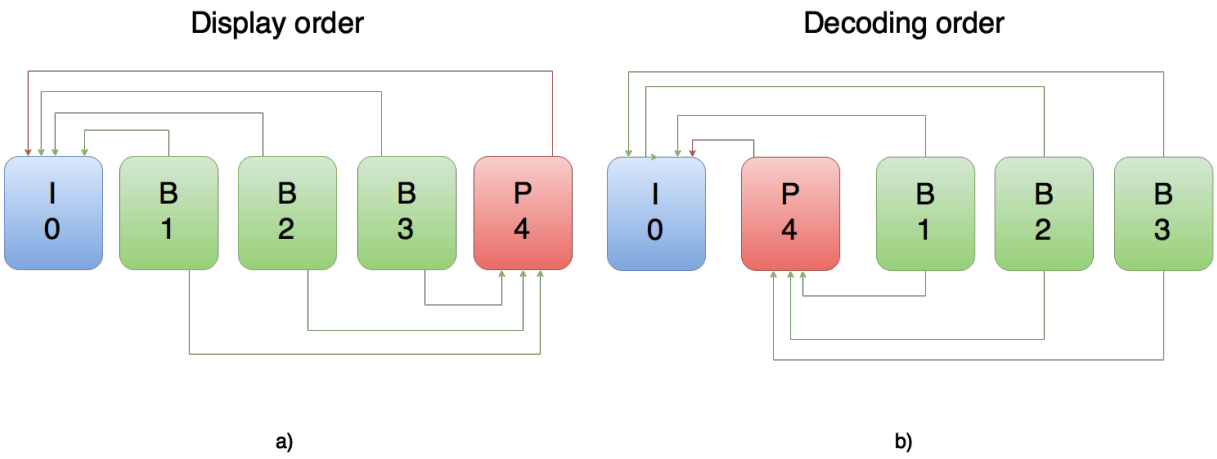



Figure 3.7: Illustration of a display order and a decode order

If our display order is the same as a) illustrated leftmost in figure 3.7, then our decoding order would also be same as b). As we know from 3.5.1, a B-frame needs both a previous frame and a future frame to specify its content. Thus I0 and P4 needs to be decoded and available to the decoder before we can start decoding B1, B2 and B3. It is possible to reorder the frames to speed up the process of decoding, but the trade-off is longer encoding time.

### 3.8.3 File and folder structure

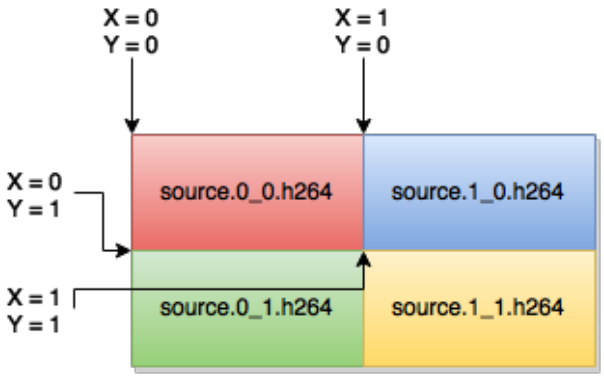



Figure 3.8: The offset used to find the surface origin[8]of each tile are added to the file name.

As we mention in somewhere, the virtual viewer in the bagadus system fetches tiles by using the index of the tiles. The file name needs to be unique and easy to identify so the virtual camera can fetch the corresponding tiles it needs. It was decided the file name of the tiles should include the source name of panorama video and then add the offset of where we found the first byte of the image from the panorama video. Figure in 3.8 illustrates the concept of how we name the files.

[8]Surface origin is the first byte of an image, also mentioned in 3.7

**Module Implementation - creating folders and sub-folders**

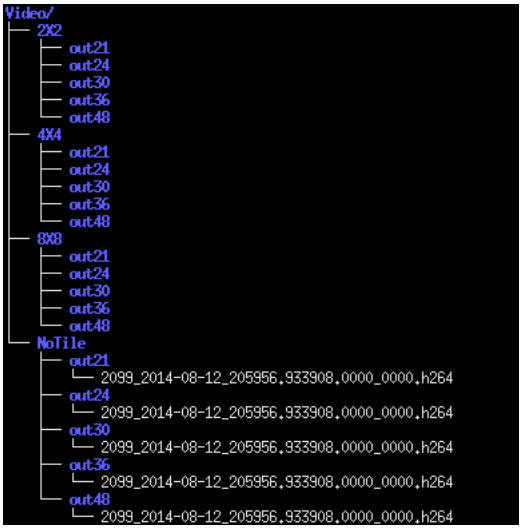



Figure 3.9: Illustration of the file and folder structure after encoding a panorama video.

This module was implemented because of the files created by encoding a panorama video to tiles in different qualities. The file names based on the concept mentioned in 3.8.3 should be unique enough to avoid overwriting of files, but there is a catch. Files which are encoded in different qualities would have the same name, since they have the same source name, the X and Y offsets. The solution was to categories each quality in their own folder. The general flow of making folders and sub-folders is as follows:

1. Create source folder
    (a) Create sub-folders with folder name describing the tiling method
    (b) For each of the sub-folders above
        i. Create a sub-folder with folder name describing which quality was used during encoding

The structure should look like in figure 3.9. In a Linux enviroment, the function mkdir() is used for making folders.

### 3.8.4 Adding task to queue

Experimenting with the first prototype we discovered there was several problems when we tried to execute it in parallel. If we look at the general flow of the program mentioned in 3.8.2, each time we read data from file, it will be processed and written to an output file. This procedure would be the main flaw when we tried to encode the file in parallel, but we had to experiment with different approach before we could come to this conclusion. The first proposition was we will decode the compressed data then subsequently encode the raw data and store it. This is to avoid the need to move the stream pointer at the beginning of the file each time a quality has been produced. As we know, before we can use a decoder we would have to initialize it by calling the avcodec_open2() where system resources are automatically allocated by ffmpeg. This procedure is also mandatory for encoding. Thus we would have to initialize all the encoders we would be using and keep them active through the whole encoding process. Testing the program with a small number of tasks gave us reasonable results, but as we increase the workload we noticed a decline in performance. Hence the weakness of this approach is the overhead produced as a consequence of initializing and maintaining several codecs simultaneously. Another approach was mentioned but it was denied rather fast. But the general concept was to give each available thread an instance of the encoder, which is the first

prototype at that time. This also means each encoder will get a decoder and an unique file. Files accessed by ffmpeg has to be distinct, as such we have to duplicate the file for each encoder. It has the same shortcomings as the first approach, but it also consume more storage space and file duplication is redundant. The reason why this module is necessary will be discussed in 3.8.4 and it is also the third approach for solving the flaw we had.

**Prototype 2 - Parallel Encoding with Tiles**

The second prototype was developed to handle some challenges we had in the first prototype. When we introduced tiling to the first prototype, the performance was inadequate. This is most likely due to the amount of work produced by the tiling module, which will be further discussed in 3.9, but basically it is a procedure where we split a video into smaller parts. Another problem that came to our attention was the CPU usage. In sequential encoding, we could not utilize the full potential of the CPU, due to the fact that we had to encode each and every video streams(the tiles) one after another in a succession fashion. Thus the solution was to utilize more of the CPU resources by introducing parallel processing. This approach should boost the performance, since we are dividing the work load to several threads. But it also introduced some new challenges, which was discussed in 3.8.4. Essentially it was about where to obtain the video stream, and the solution we came up with was to decode a panorama video and store the frames in memory before encoding. The second issue, which was not mentioned, was synchronization between the threads. The problem we had was how to identify which part of the video has been processed and which has not, and also keep track on which quality has already been encoded. The easiest solution we had at the time was to have the parent thread keep track of every operations. Whenever a child thread is created by the parent thread, it will get two values. One is the tile number which is used for calculating positions(further discussed in 3.9 of how it works) and the other is a value which represent the quality the tile has to be encoded with. Without going into further detail, since this is a prototype, the general flow of the encoding after video files had been preprocessed and stored in memory:

1. For each tile or tiles and their qualities it will
    (i) Create a child thread and during the its life cycle it will
        (a) Fetch a task from queue
        (b) Calculate the position of the surface origin of the tile
        (c) Initialize an encoder with a given CRF value[9]
        (d) For as long as there are still frames left
            A. Fetch a frame from memory
            B. Make a replica of the tile by copying raw yuv data from a given area in the frame
            C. Encode the yuv data and write it to file
        (e) Close encoder
    (ii) Terminate child thread

**Adding task to queue - Module implementation**

The third approach was to split the program into two parts, the first one will pre-process the files and make them available in memory. The second part is encoding and storing the outputs to their respective locations in the folder structure. This was heavily influenced by the reason for storing files in memory as mentioned in 3.8.2, but there are still some challenges that needed to be resolved. We know from the initial testing of the second prototype, that tiling and encoding of a panorama video is very well suited in a parallel enviroment. Even though we cleared up some of the obstacles, it still suffered from considerable overhead. This can

[9]Constant Rate Factor, mentioned in **??**

been seen by increasing the amount of tiles it has to encode. With more tiles it has to initialize and maintain more encoders thus overhead is produced. Another cause is the sheer amount of context switching between the threads, and lastly the creation and termination of threads which is also very costly. Thus the solution we came up was to create and maintain a given number of threads. The difference is when they have completed their task, they will remain idle and wait for new tasks instead of getting terminated. Hence we avoid the unnecessary overhead produced from the previous approach.

In the second prototype we had a thread act as an overseer to synchronize the whole encoding process of every threads, therefore we did not fully use the CPU. Thus we decided to also use the parent thread for encoding. That is why we needed a structure where every thread can execute independently without the need to be supervised by another thread, or as little as possible. That is why we implemented the Task Structure. It has 2 main functions, addTaskToQueue() and getTask(). As the name implies, the first function adds task to queue, but it also has the responsibility for creation of tasks that are mandatory for synchronization between threads. The information stored in a task structure will be used by a thread to identity which tile to process, what quality it should be encoded with and where it is going to be stored. Thus threads can fetch a task with getTask() from the queue and operate independently. Since we are executing the application in parallel, there can be cases where two concurrent processes gets the same task. To prevent a race condition[10] from happening we added a mutual exclusion object(mutex)[11] to ensure that every threads can access the queue but not simultaneously, thus every threads will get unique tasks. The general flow of addTaskToQueue() is shown below:

1. Extract the source file name from the source

2. For each tile, do the following:

   (i) Calculate the x and y position of a tile based on which tile or tile number it is

   (ii) For each quality we have to encode a tile with, do the following:

      (a) Find the coordinates of the surface origin of the tile in panorama video

      (b) Create an AVCodecContext [12] with a given quality

      (c) Create an output file name with the folder address(based on the number of tiles) as a prefix, then append the source name with the coordinates we calculated previously

      (d) Create a Task Structure, then store the tile number, AVCodecContext and the output file name.

      (e) Push the Task in queue

[10] A situation where several processes or threads access a shared data and perform operations on it at the same time, thus the output will vary according to which operation was executed first, which is undesirable in most situations

[11] mutual exclusion object or short for mutex, is a program object that allows multiple program threads to share the same resource, but not simultaneously (35)

[12] Creation of this type of structure would not cause overhead, it is when initializing that we would require the use of system resources.

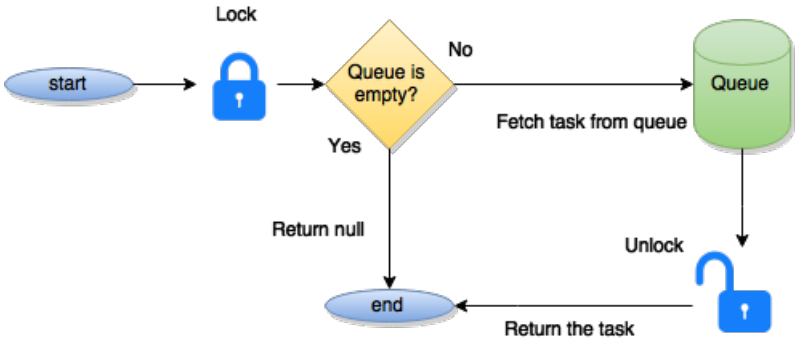



Figure 3.10: Illustration of the flow of getTask() function.

We can see the flow of getTask() in figure 3.10. When called upon the function will lock itself to prevent other threads from accessing the queue. Then it will check if the queue is empty, a task will be obtained from queue if there are still elements in it and the data is then returned to the callee, else it will go back with nothing. Before returning, the lock will be freed and the next thread in line can access the function.

## 3.9 Tiling module

In this thesis, a tile is a sequence of frames which are extracted from a specific area of a panorama video and then encoded into a video stream and stored as a raw H.264 file format using x264. Thus tiling of a panorama video means that we are separating the video into smaller video streams where each of them has equivalent resolutions. These tiles are processed in such a way that they would not overlap with each other when we create the original panorama video by stitching the tiles together. The tiling module main responsibility is to separate a panorama video stream into equivalent tile sizes.

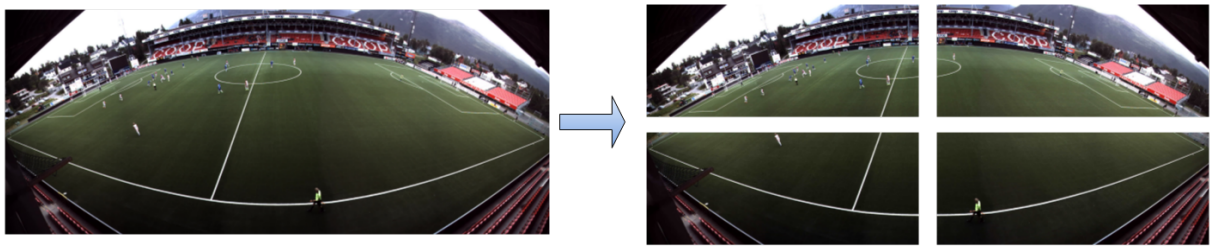


Figure 3.11: Separation of a panorama to 2X2 tiles

### 3.9.1 Tiling formations

In the thesis there was 3 tiling formations that was proposed, 2X2, 4X4 and 8X8 tiling. The first number before X determine how many tiles are created in the horizontal direction, and the next number decides how many tiles we are going to have in the vertical direction. There were other formations that was mentioned, such as 16X14, 9x9 3x3 and so on. But there were either too many video streams created(1120 for 16X14) or the extra calculations and padding as a consequence of tiles with odd numbered resolutions being created. Thus it was decided to use 2X2, 4X4 and 8X8 tiling approaches for the panorama video stream with a resolution 4096x1680.

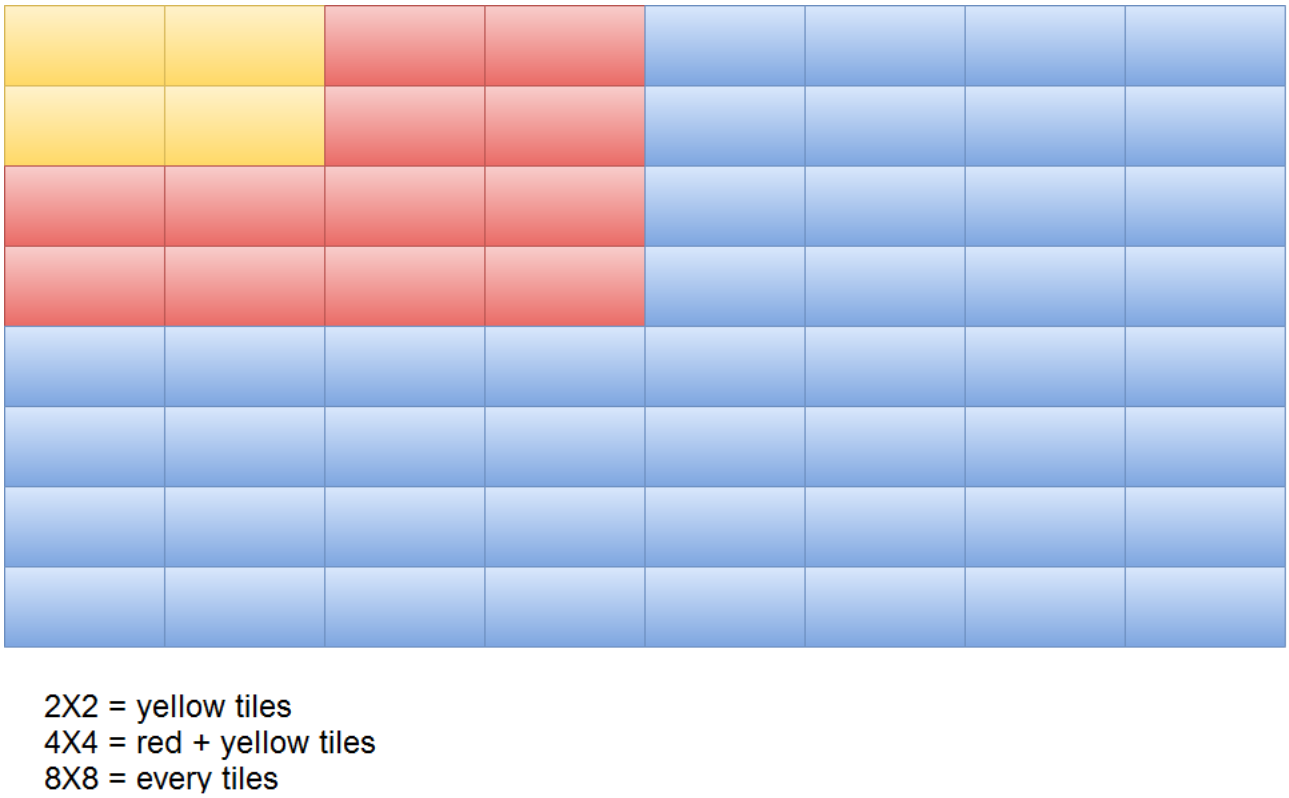



Figure 3.12: Overview of the different tiling formations

### 3.9.2 Image processing

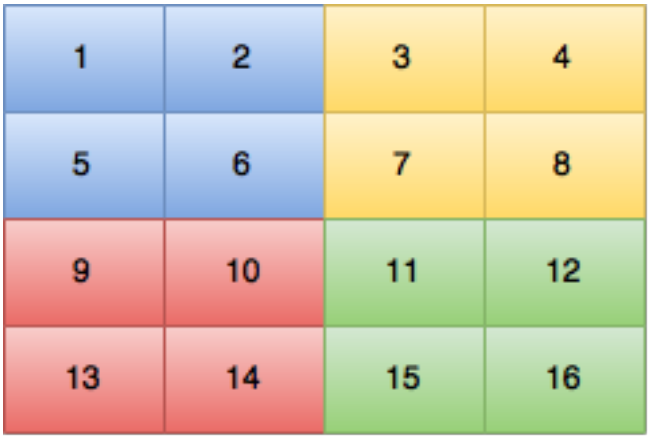



Figure 3.13: How pixels are represented on an image

A tile can be described as a subset of tiles which together makes a whole panorama video. But the problem is how to separate a panorama to tiles. We have mentioned in 3.7 that a video stream is a sequence of images. So to create a tiled video stream means we have to go through every frame and extract the corresponding area. An image can be described as rows of pixels, where each pixels are numbers indicating the variations of red, green and blue. As a whole it will provide a representation of a picture. Everything on screen is made of pixels, and we are familiar with the idea that each pixels have an x and y position on a two dimensional window. An example is drawing programs, they usually has a window or corner where they show the x and y position of the pixel your mouse is pointing on. However, images stored in memory are a linear sequence of color values, thus only one dimension.


Figure 3.14: How pixels are represented in memory

When we are transcoding with no regard of which pixels we need to have, there would not be any problem. Since we can access each pixels one by one and copy the data over to a new frame. If a panorama video is represented by every pixels as shown in figure 3.13, then each tile in a 2X2 tiling approach would be represented by each grid(with 4 pixels) in different colors. The image stored in memory would look like figure 3.14, thus the pixels which represent a tile cannot be extracted in a sequential order. In order to successfully make a tiling system, we have to be able to extract the pixels which belongs to a specific tile. At the initial phase a formula was proposed by a colleaguewhich could find the first pixel of every tile on a panorama video. The process can be described as follows:

1. Assume an image where the number of tiles is n in both vertical and horizontal directions
2. Find the x position of a tile with:

   tile number % n
3. Find the y position of a tile with:

   tile number / n
4. The first pixel of a particular tile can be calculated with:

   (x * tile width * n) + ((y * tile width * n) * height of the panorama video)

These steps made it possible to find the first pixel of a tile in memory. Even though it worked, there was a limitation. The restraint we found was; it does not work if the tile formation does not have the same number of tiles horizontally and vertically. Thus we did some research and made some improvements to the first approach to overcome the second restraint.

1. Assume an image has n number of tiles in the horizontal direction and m number of tiles in the vertical direction
2. Find the x position of a tile with:

   tile number % n
3. Find the y position of a tile with:

   tile number / m
4. The first pixel of a particular tile can be calculated with:

   (x * tile width) + (y * tile height) * panorama width

After testing it could handle the tile formations we used in the thesis and also compatible with tile approaches with different horizontal and vertical sizes. The reason it works is because we exploits the fact the image in memory is expressed as a one dimensional array, thus the first pixel is in index 0. So if we make some adjustments to our numbering of tiles, such as tile number 1 is expressed as 0 and tile 2 as 1 and so on. Then update the memory layout figure to represent the index of the pixels in an array. An example of how the whole process works is shown in figure **??**.

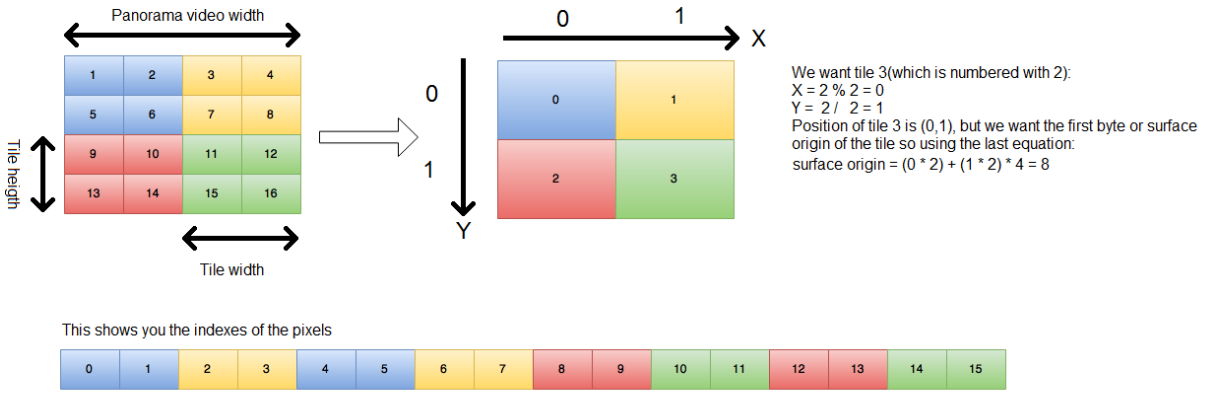



Figure 3.15: Example of how to find the surface origin of a tile

How to copy every pixels of a tile is covered in 3.11, since the calculations can vary in accordance to the pixel format we are using.

## 3.10 Encoding concepts

There were several concepts of how to encode the different tiling formations we have. We expect the some of the concepts would not give us any satisfactory results, but there are many unknown factors to us which can be revealed by implementing and testing every approaches. The knowledge gained from the experiments can be used to determine the origin of overheads, bottlenecks, bugs and many others. Thus we can make adjustments to the implementation to correct the problems and at the same time improve our design.

### 3.10.1 Sequential Encoding Concept

At the initial phase of the project, we needed to implement a prototype for experimentation. Thus we adapted the idea of sequential programming[21] to our scheme for its simplicity.

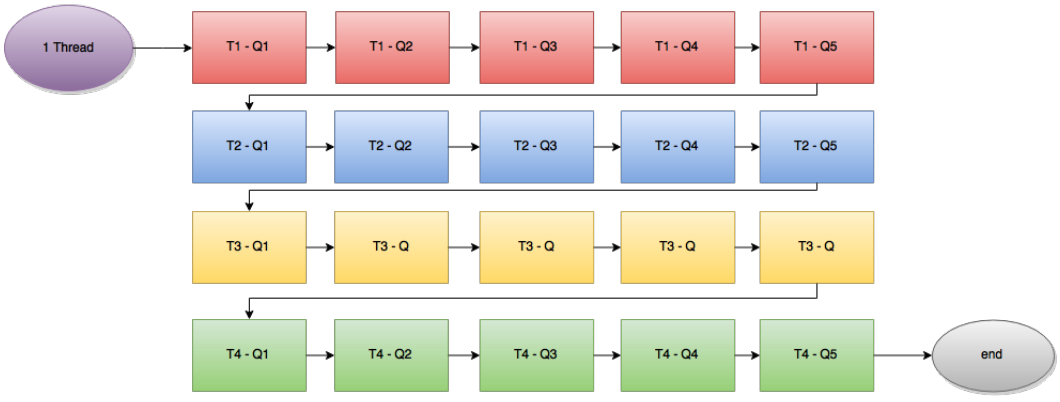



Figure 3.16: Overview of sequential encoding

The figure 3.16, gives us an overview of what happens when we encode a 2X2 Tiles into different qualities sequentially. At the beginning we will encode the first tile and when the process has been completed, it will proceed to encode the next tile. This procedure will continue until every tile with different qualities are encoded. Since we are using only 1 thread to encode every tile, we would not be able to utilize the CPU efficiently. Thus it is very probable that the results from sequential encoding would deviate substantially from the outcome of both parallel and thread pool encoding.

FFmpeg has an optimization procedure called frame slicing, where a frame is separated into smaller parts, and each part will be processed by a thread which are created internally of an encoder. Thus we will also test the sequential encoding concept with frame slicing to see if there are any huge performance leap. Maybe it could even be able to satisfy our design.

### 3.10.2 Parallel Tiling Concept

One of the main goal in the thesis is to separate a panorama video into smaller equivalent video sizes, and encode each of them in multiple qualities. This produces a lot of work which will probably lead to poor performance with the sequential approach. Thus we introduced another approach adopting the idea of parallel computing, where we can execute multiple operations simultaneously. There were 2 ways to decide the number of threads. The first one is shown in figure 3.17 and the second in figure 3.18.

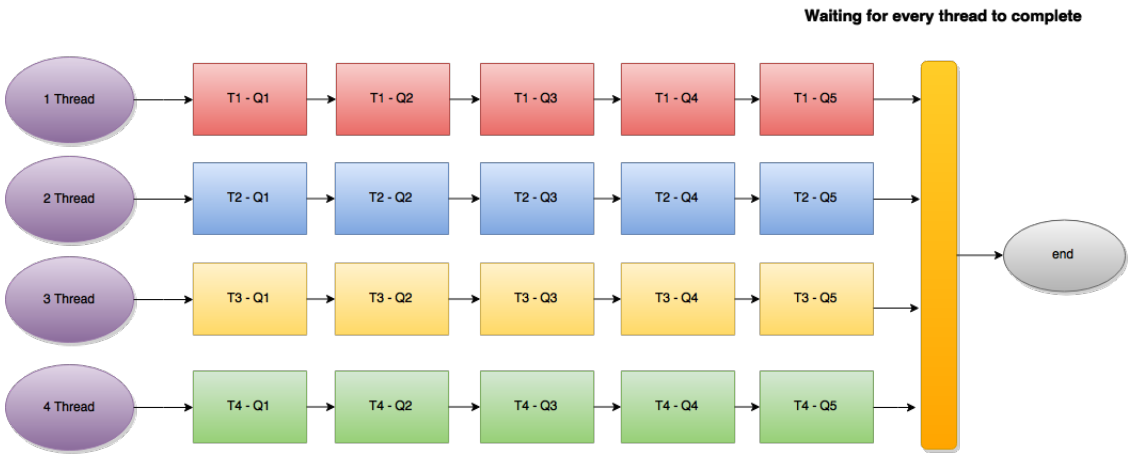



Figure 3.17: Overview of parallel encoding. The number of threads created are the same as total amount of tiles

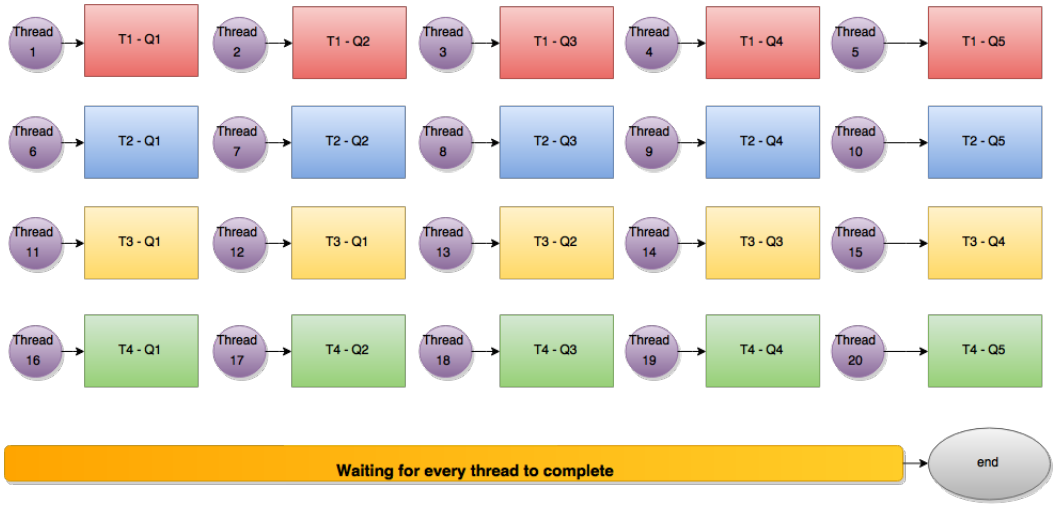



Figure 3.18: Overview of Parallel Encoding. The number of threads created are the same as total amount of tasks.

We know there is overhead produced whenever a thread is created and terminated, but how much does it affect our implementation. So instead of randomly picking a number of threads to tests all the different tiling approaches, we decided to use the number of tiles in accordance to the tiling approach used to determine the number of threads being created in the first parallel concept. For the second concept the number of threads are decided by the total amount of video streams that is going to be created. Thus for 2X2, we will only get a maximum of 4 threads working at the same time on the first concept, each of them will be created and terminated whenever a stream is created and completed. There will be 20 threads working simultaneous in the second concept, both approaches uses 20 threads during its lifetime.

### 3.10.3 Threadpool Tiling Concept

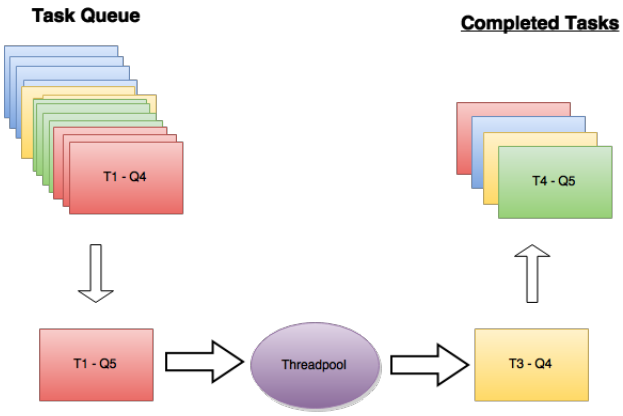



Figure 3.19: Overview of Threadpool Encoding. The number of threads will be decided according to the current hardware,

Thread pool is a group of threads that has been created and put in a standby mode, waiting for jobs. When a task or operations that needs to be executed, it will be transferred to the pool where one of the pre-initiated thread will work on the task. There are many implementation of thread pool which can be found online and can easily be integrated in our program. We decided to not use the implementation but only adopt the concepts of thread pool as shown in figure 3.19. We will also create threads at the beginning and let them be idle, but we will not use them until we have filled up a queue. This is to get a more precise evaluation of the concept,

since if they work during insertion of tasks to a queue and a task only needs a few operations to completed. The other threads would not have the chance to work. This restriction is due to the fact that we are working on video files stored locally and they are in H.264 format, thus we have to read and decode them before we can process them. This procedure also requires resources from the system, hence the limitation. The performance with the use of thread pool should be better than sequential and parallel concepts. We expect the number of threads in a pool which gives the least encoding latency should correlate with the number of physical cores in a CPU, since video encoding are CPU-bounded.

## 3.11 Encoding pipeline implementation

The encoding pipeline has the responsibility to separate a panorama video into n x n tiles, the n stands for the number of tiles. Each tile is then encoded into multiple qualities and stored. The end of our software encoder should look something like 3.20

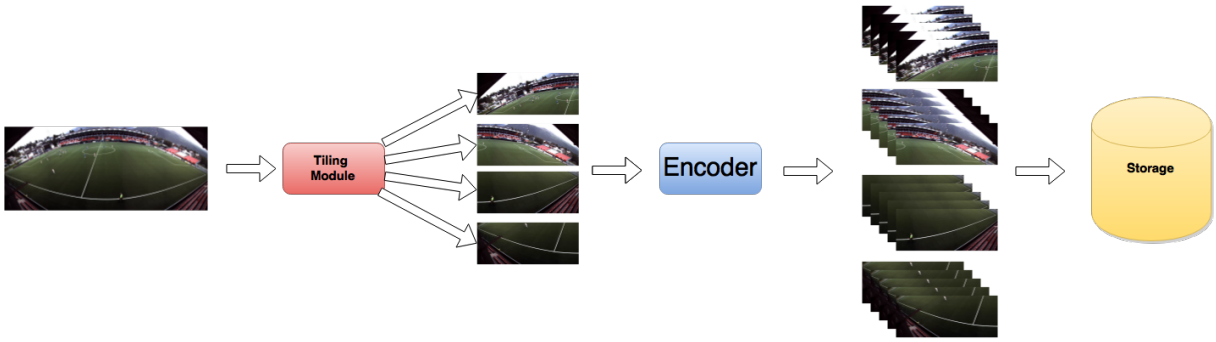



Figure 3.20: Overview of the encoding pipeline

### 3.11.1 Initialization

The encoding process starts upon calling the function startEncoding() in our software encoder implementation. As we know from 3.3, the file formats we are using to store files are h.264 because of its efficient compression while retaining good quality. Those are a standard which describes how the video stream is compressed so it can be decoded during playback. Thus we need a software encoder which encodes a video stream into h.264. There are many encoders available that can encode video streams into h264 as mentioned in3.4, but we chose x264 for its efficient compression capability while retaining good video quality and its flexibility. As mentioned in **??**, we are going to use the libraries in ffmpeg for encoding. Before we can proceed to encoding of a video stream, we have to find the corresponding encoder by using avcodec_find_encoder_by_name() which locates a registered encoder in libavcodec with the a specified name. If an encoder is found we will allocate memory for a new AVFrame with getDummyFrame(), the information in it is based on the number of tiles.

### 3.11.2 Allocation of an AVFrame Structure

During demultiplexing and decoding we also used AVFrame, but the AVFrame we created was only a shell without any data of a specific frame inside it. The AVFrame is allocated and filled by ffmpeg automatically during decoding. However in encoding, ffmpeg do not have the necessary information to execute the same procedure, thus it has to be manually done by us. Though we could directly encode the AVFrames(decoded from source) stored in memory to new video streams with different qualities, but that would counteract the purpose of this thesis. Hence the implementation of getDummyFrame(). The function getDummyFrame() can be described in these steps:

1. Create an AVFrame with av_frame_alloc()

2. Set the pixel format of frame given by callee

3. Set width and height of frame given by callee

4. Allocate an image to AVFrame with av_image_alloc()

5. Return the AVFrame

As you can see, we allocated twice for an AVFrame with 2 different functions. The reason for this because ffmpeg does not have sufficient information to determine how much memory to allocate image data, thus the first function only allocates a space for a generic AVFrame and set it fields to default values. As we know from **??**, an image may consist of several channels according to the pixel format, and these channels are stored in AVFrame as uint8_t data pointer or pointers(depending on the pixel format). When the width, height and the pixel format is known, then we can use the second function to allocate a memory space for an image and populate the AVCodecContext::data pointers. The function will also calculate the stride of each channel and store it in AVCodecContext::linesize field.

The reason for getDummyFrame() is to get an AVFrame which can act as temporary storage for a frame of a tile, thus we needed to manually allocate the AVFrame corresponding to the resolution of the tile(and pixel format if there is a need for conversion). There should not be any significant overhead caused by getDummyFrame() since it will be executed only once during the whole encoding life cycle and the frame will be reused by overwriting the old data with new information.

### 3.11.3 Encoder settings

When the queue has been populated with tasks and we have initiated a number of threads for encoding(depending on which encoding concept we are using), each of the threads will be autonomous until every tasks has been completed. At the start a thread will attempt to fetch a task through getTask(), depending on the result, it will either proceed to encoding or remain dormant until there are new tasks. When a task has been given to the thread it will initiate the libx264 encoder from ffmpeg and initialized it with an AVCodecContext from Task structure.

An AVCodecContext is a structure used to control various properties of an encoder. Many of the settings that we are using are predetermined beforehand to work with the client side implementation of Interactive Virtual Camera, but I will run some test on on various settings to find the pros and cons of using them, and find out if there are any way to further improve the design. Prior to using the AVCodecContext strucure, we have to find out which encoder the context is going to be used for, thus we have to call avcodec_find_encoder_by_name() with libx264 as the input. Then we can allocate the structure with avcodec_alloc_context3() with the given encoder and populate it with the settings we want. The predetermined option is as follows:

AVCodecContext data fields

| | |
|---|---|
| gop_size | 90 |
| refs | 1 |
| time_base.den | 30 |
| time_base.num | 1 |
| ticks_per_frame | 1 |
| max_b_frames | 3 |

Table 3.2: These settings was determined by Vamsii who worked with the clients side of Interactive virtual camera

1. A short explanation of the data fields used

(a) gop_size refers to the size of a Group of Pictures.

Advantages and disadvantages are discussed in 3.5.2

(b) refs = how many frames can be used as a reference frame

Advantage: In H.264, using several frames as a reference frame can improve compression efficiency

Disadvantage: Multiple reference frame can substantially increase encoding time

Disadvantage: Considerable increase in memory usage during decoding, since reference frames must be stored until they are no longer needed

(c) max_b_frames Maximum number of B frames in a GoP

**gop_size**

gop_size is a limit the we can set to tell the encoder the maximum size of a Group of Pictures. The advantages and disadvantages are discussed in 3.5.2 and seen in figure 3.1.

**refs**

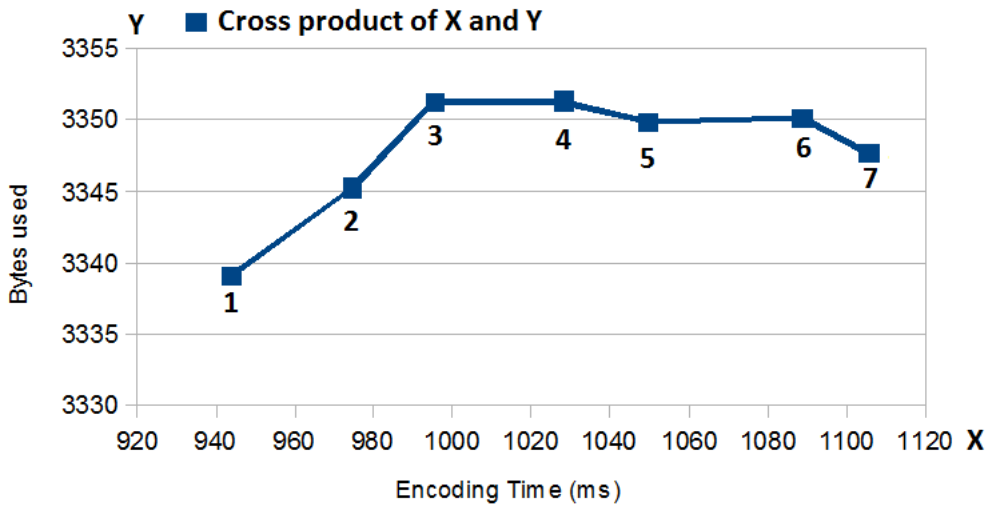



Figure 3.21: Correlation of between encoding latency and compression efficiency. The number under the squares represent the amount of reference frames we used.

refs refer to the number of reference frame an encoder can search for redundancies while encoding. We have mentioned frames which can be a reference frame and used by other frames to fetch information from in 3.5.1. In H.264, P-frames can have more than one previously decoded reference frame during decoding, and B-frames can also use more than 1 previously decoded frames as reference. The same concepts of using reference frame to have more efficient video compression rate in 3.5.1 should apply here too. Extra reference frame should lower the bandwidth, but increase the encoding latency since we have to search after more frame to use as a reference. Though if we observe the figure 3.21, the bandwidth reduction does not seem to be in our favour in this case. As we can see, the bandwidth increased in accordance to the number of reference frames we used. The increase in encoding time was in our expectation. We do not see a positive behaviour until the number of reference frame is set to 4, that is when the consumption of the storage is starting to decrease. The results may have been affected by our own settings. The duration of our video is 3 seconds, and the FPS is 30 thus we have 90 frames. We also set the maximum of B-frames to 3, hence we can only have max 3 B-frames which can come subsequently after each other. To summarize, we have frames that has been used as a reference frame already, and by adding more, the information of where the new reference

frame is has to also be stored thus the increasing bit rate. However, after increase the number of reference frame beyond 3, the bitrate begins to decrease. The reason is currently unknown, but since setting the AVCodecContext::refs to 1 gives us the least encoding latency, we will be using it. This just proves to us not every encoding parameters are compatible, thus we need to test and evaluate the different settings.

#### Determining the frame per second(FPS)

Frames Per Seconds, or FPS, is used to determine the rate of frames that are shown during playback, it is compulsory to have it. In FFmpeg, the FPS is calculated by a nominator and denominator which represents a tick in one second and apply how many ticks to represent a frame. We had a denominator set as 30 and nominator set as 1, thus 1 tick is 1/30 of a second. Consequently, it means 1 seconds has 30 ticks and since ticks_per_frame are set to 1 we will have 30 frames per seconds. The reason for this approach is because in some codecs, the time base is closer to the field rate than the frame rate. In field rate, a video frame can be comprised of two fields. The fields use interlacing between still images to simulate motions. But we will not go into further detail since this is not the focus here. To summarize, FPS are represented differently in accordance of codec thus the need for calculations.

#### max_b_frames

An encoding parameter which are used to tell the encoder the maximum amount of B frames which can come subsequently after each other. We discussed the difference between picture types in 3.5.1.

#### Dynamic settings: Resolution, pixel format and thread_count

What we mean with dynamic settings are parameters that varies according to which approach we are using. Such as in 2X2, the resolution of the tiles are 2048x840. Pixel format refers to which color space or variations of it we want the stream to be encoded in, changing pixel format could potentially reduce bandwidth. These two parameters are mandatory since the encoder need to have knowledge of the width and height of a frame, and pixel format to do correct calculations for compression during encoding.

The encoding parameter thread_count refers to how many child threads can be used during encoding. The numbering decides whether FFmpeg will activate frame slicing which will reduce encoding latency. Basically it will split a frame up into separate smaller frames, then create and dedicate one child thread to each slice. Setting thread_count to 0 will give FFmpeg the freedom to choose how many child threads it will create during encoding, and 1 encodes normally without any optimization.

### 3.11.4 Choosing a preset

A preset is a pre-defined collection of encoding options that are designed to accommodate one specific type of scenario. Thus encoders usually provide a set of default configurations for various settings. The presets can also be used as a base then the user can add their own encoding configurations which will override some or all of the presets encoding parameters if they collide. We tested the following default configuration in x264 with our encoding parameters from 3.11.3.

| Preset used | Encoding Time (ms) | Size of the output file (bytes) |
|---|---|---|
| *ultrafast* | 957.79 | 3419191 |
| *superfast* | 1245.11 | 3050601 |
| *veryfast* | 1182.17 | 518973 |
| *faster* | 1756.50 | 570575 |
| *fast* | 2083.88 | 817215 |
| *medium* | 2683.02 | 771893 |
| *slow* | 3083.80 | 782174 |
| *slower* | 4351.90 | 727461 |
| *veryslow* | 4942.89 | 611973 |
| *placebo* | 27208.39 | 667652 |

Table 3.3: Comparing x264 preset in combination with our encoding parameters. Encoding of a panorama video to 1 quality

We optimally want very low encoding latency, high compression rate while having high video quality, but unfortunately it is still not possible in real-life. Based on the results from table 3.3, it seems like the latency increases in descending order, but the size of the file decreases. According to FFmpeg[58], the superfast configuration should be faster than veryfast, and the file sizes should decrease following the order. This behaviour is most likely due to the collision of encoding parameters set by us and the presets. The configurations that are relevant for this thesis are those designed to decrease the latency, faster thus better. ultrafast came out as the top in encoding time, but the file size was also the largest. The composition of encoding parameters in veryfast was not so far behind and the output was compressed very efficiently. As such we cannot decide the preset to pick. since we do not have sufficient information to know how much the overhead from tiling could affect these presets. Thus when the design and the implementation has been finalized, we will experiment with the two default configuration again to get a more complete evaluation. The only way to set the preset in x264, is through av_opt_set() which is a function that sets parameters which are not native or found in a generic AVCodecContext structure.

### 3.11.5 Setting CRF value

We are using CRf as rate control, thus the file size would be unknown until the file has been created. Setting of CRF value follows the same step as presets. By using the av_opt_set() to explicitly tell the x264. The value for each tile but the values we are using for the tiles are: 21, 24, 30, 36 and 48.

### 3.11.6 Storing the media

When a task has been fetched from getTask() and the encoder has been initialized in accordance to the encoding parameters stored in the task. The next step is to define a storage medium to store the encoded bitstream. There were 3 approaches we tried:

1. Store the encoded bitstream in memory
2. Write it to file through FFmpeg
3. Write it to a blank file

Storing the encoded bitstream in memory should not be a problem with our machine configurations(8GB ram), but we have to keep track of every encoded bitstreams from each tile in each quality(8X8 tiles will give us 320 video streams). There are maintaining to do such as allocating and deallocation of memory, reference pointers, synchronization if it is shareable

and some other. Nevertheless, we would need to store the encoded bitstreams to disk after the encoding sessions.

Writing to file was another approach, and FFmpeg has its own structures and function for IO operations on a file. For each file, FFmpeg would need to create a structure to store information. But before we can use them(IO operations), like many other functions from FFmpeg, we would need to initialize them thus requires system resources. Keeping them open over a period of time would further increase the requirement. Closing them would release the resources, but then we would not be able to access them before re-initialization of it. Testing with 2X2 approach did not have any noticeable performance decrease, but the more tiles we used the worse it performs.

The third approach was to write the encoded data into a blank file through fstream[16]. We discovered later that it was possible to create a blank file and write a video stream to it directly. Usually a stream header is needed so an application can read and identify how it is coded to decode it correctly. But we found out later it was during the encoding of the first output bitstream that a stream header is created and included. Some tests were made with different tiling approaches, one where we left the files open and another where we will only open them, write to files and close them. We noticed there were lower resource usage with the second test but the amount of IO calls it made a little affect on the performance. The CPU would be set to wait state while writing to disk. Even though with a SSD, every calls are executed at a fast rate, the build up after a period of time would still be noticeable. Thus we decided to use the third approach, avoiding the extra structures and functions required, and let the files be open even though it used more resource.

### 3.11.7 Processing and encoding of a tile

When a file has been created, then we will proceed to extracting a tile from a panorama video stream. We have make some calculation to find the surface origin of the tile. As we know from 3.11.2, we used getDummyFrame to get an AVFrame structure and it is used to used as a temporary storage during the encoding process. We have mentioned before that the frames from the panorama video stream are stored in memory as AVFrame structures. The way it stores changes according to which pixel format, but for YUV 4:2:2, the image data(Y, U and V channels) are stored and referenced by AVFrame::data pointers, and their strides(width+padding if any) are stored in AVFrame::linesize. To find the surface origin of a frame we can calculate it with equation which was created and discussed in 3.9.2

$$surface\ origin = (x * tile\ width) + (y * tile\ heigth) * panorama\ width \tag{3.1}$$

We are exploiting the fact that the YUV channels has been stored as a 1 dimensional array, thus we can adopt the same concept from image processing here. We know Y channel is never sampled, so it has the same resolution as the image we look at. Thus the position we found through the formula can be used as an offset to the first byte of the Y channel. The offsets of U and V is always half of the offset of Y, because U and V is sampled half the rate of Y horizontally in YUV 4:2:2. Consequently, it also means that the height are the same between Y and UV. The height are used to keep track of which row we are in when tiling. Thus we can fetch the pixels from all three channels at the same time. Getting to the next row can be done by incrementing the offset of Y by 2, and for U and V divide Y by 2 to find their offsets. Each time we find the offset of all three channels, we copy the size of tile width data to the Y channel. With U and V only the amount of data to be copied is half of the tile width. Transferring data from panorama to a tile can be seen in figure **??**

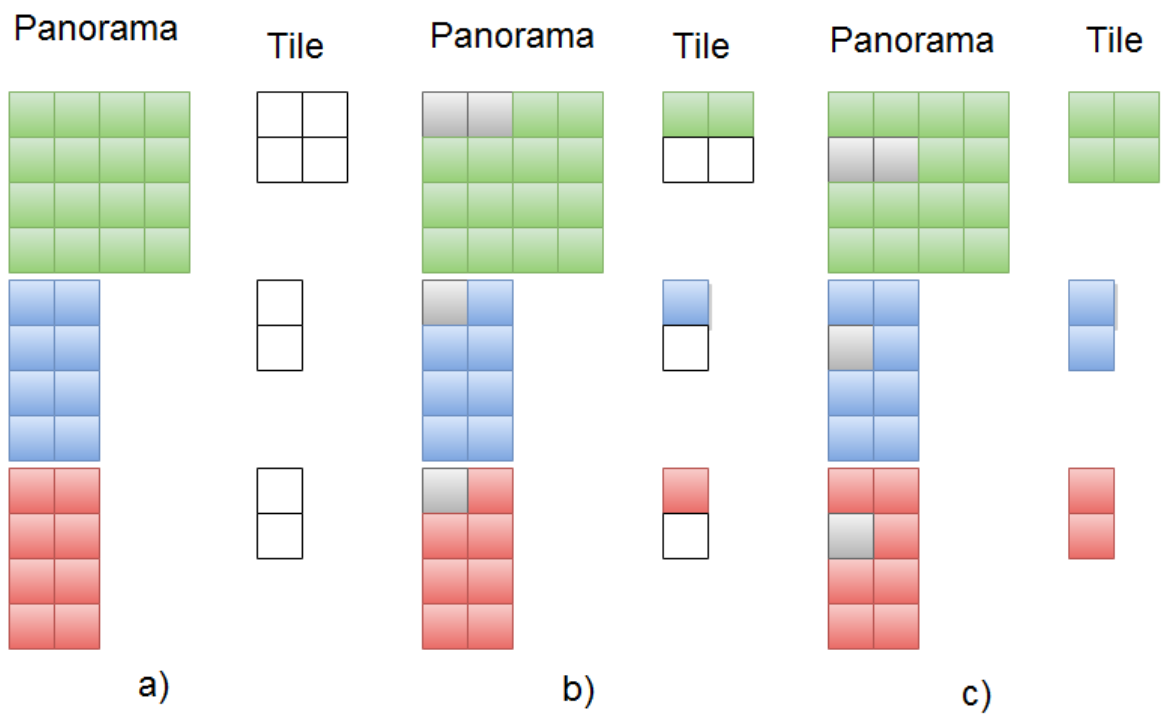



Figure 3.22: Overview of how to transfer data from panorama to tile

a) is the initial state of the tile. When the Y, U and V offset has been calculated, copy the pixels at the offsets corresponding to the channels b). The amount of pixels to be copied is always the width or stride of a channel, so from b) it was 2 pixels and from U and V only 1. To get to the bottom row of the tile, we have to add the width of the panorama, thus the Y offset will land on the first grey area on the next row and replicate the same procedure as b), then we get c).

We have only described the procedure to get the right information for a tile, to produce a tiled video stream can be summarized in these steps:

1. Find the surface origin of the tile, which is also the offset of $Y_{tile}$
2. If there are still frames left, do the following:
   (a) Fetch a $frame_{panorama}$
   (b) If there are still rows left in the $frame_{panorama}$
      i. Copy stride of $Y_{tile}$ worth of data from $Y_{panorama}$ channel based on $Y_{offset}$
      ii. Copy stride of $U_{tile}$ worth of data from $U_{panorama}$ channel $Y_{offset}/2$
      iii. Copy stride of $V_{tile}$ worth of data from $V_{panorama}$ channel based on $Y_{offset}/2$
      iv. Increment $Y_{offset}$ with the width of $frame_{panorama}$
   (c) Write the presentation time stamp to $frame_{tile}$
   (d) Encode the $frame_{tile}$
   (e) Write the encoded bitstream to file
3. Flush the encoder and write the remaining encoded bitstream to file
4. Close the file

## 3.12 Summary

In this chapter we have presented different approaches to reduce the storage of files thus reducing the bandwidth consumption. We have discussed the h.264 codec and its encoder x264(libx264 in ffmpeg) and how we can affect the video quality by modifying the options available in x264. We have gone into detail of how our design was constructed through simple steps. First we designed a simple design and implemented it, through experiments with the prototypes we discovered different ways to improve many aspects of the program thus removing unnecessary overhead. We have mentioned the process of how to make a tile of a panorama video and encode them. Some small tests were conducted on encode settings for verification and finding the best option to use for further testing the final design and implementation of our thesis.

# Chapter 4

# Case Study: Software Encoding with libx264

In this chapter we will be presenting the results of our approaches and their implementation. Furthermore we will discuss the overhead produced with these designs and compare them to each other. We will also explore the advantages and disadvantages between the tiling approaches. The experiments will reveal to us any correlation between the level of compression and encoding latency and their consumption of storage space thus bandwidth.

## 4.1 How experiments are done and measured

### 4.1.1 Set-up specifications

The machine we used conducting experiments and performance measurement are equipped with an Intel Core i7-4700, 8GB RAM and a SSD hardisk. The configuration we used for FFmpeg can be found in the Appendix 1. The panorama video we are processing has a resolution of 4096x1680 and YUV422() as pixel format.

### 4.1.2 What kind of experiments are conducted and why?

The experiments conducted in this chapter reflects on the prototypes we made during our master thesis. The prototypes are made and tested so we can rework our design and adjust the implementation to optimize it further. The duration of the panorama video are 3 seconds, and in each test the video stream will be separated into tiles according to which approach is used, then each tile will be encoded into 5 different qualities. Most of the graphs will have a red line, which is the threshold for real-time encoding. The CPU and memory usage will also be taken into account during the tests. The CPU and memory usage measurement are extracted from the system tool "top". The command used are:

1. top -d 1 -b > process.txt
2. more process.txt | grep Encode | cut -c 42-48

The first command will execute top and run indefinitely. For each second it will log the information shown in top to a text file. When the encoding has been completed we can CTRL-C to interrupt the program and causing it to abort. The second command shows the log file in terminal, only the information which has "Encode" in it(which is our process) and cut out all other unnecessary information.

Note that we are using an intel 4700 processor for testing, it has 4 physical cores and has Hyper-Technology(HTT)[1], thus it will be recognized by the operative system as a CPU with

[1]Hyper-Threading Technology is Intel's proprietary simultaneous multithreading (SMT) implementation used to improve parallelization on x86 microprocessors[24].

8 cores. So when executing top, the CPU usage percentage can vary from around 100% to a maximum of 800% during encoding. Where 100% correspond to full utilization of 1 core and 800% for all cores. We want to evaluate the encoder, not the decoder. Thus when the CPU utilization has been stable in a period of time, which means encoding of panorama stream to tiles is in execution, it is then we will take the measurement. Another way to easier differentiate the decoding and encoding part is to put the program in sleep mode right before encoding. Then start and log the CPU utilization.

## 4.2 Sequential Encoding

Sequential encoding was the first approach that we designed and implemented. It was also our first prototype to be used for testing. The origin of this idea was from sequential processing, in which every instruction is executed following in a logical order or sequence. Thus the encoding process of the first prototype is based on the mentioned approach. We doubt sequential encoding would give any better results over other approaches that we designed, but the reason for testing is to verify for ourselves and prove that the design needs to be reworked. Thus the results from this approach would be used as hints of how to optimize our design further. Furthermore we are using ffmpeg for encoding, more precisely its library to encode video streams with x264, and as discussed in they have a feature to optimize the encoding process of a frame by utilizing more threads on it. There are two experiments that could be conducted with the sequential encoding design. The first one is tested with the proposition we had, and the second test will be conducted using ffmpeg native optimization for encoding. Basically, ffmpeg can spawn new child threads and assign them independent tasks on a frame. The amount of threads can be modified(through AVCodecContext::thread_count), but after some testing we concluded that the best solution was to have ffmpeg spawn the number of threads at its discretion(.

### 4.2.1 Experiment 1: Sequential encoding of panorama video and tiles into multiple qualities

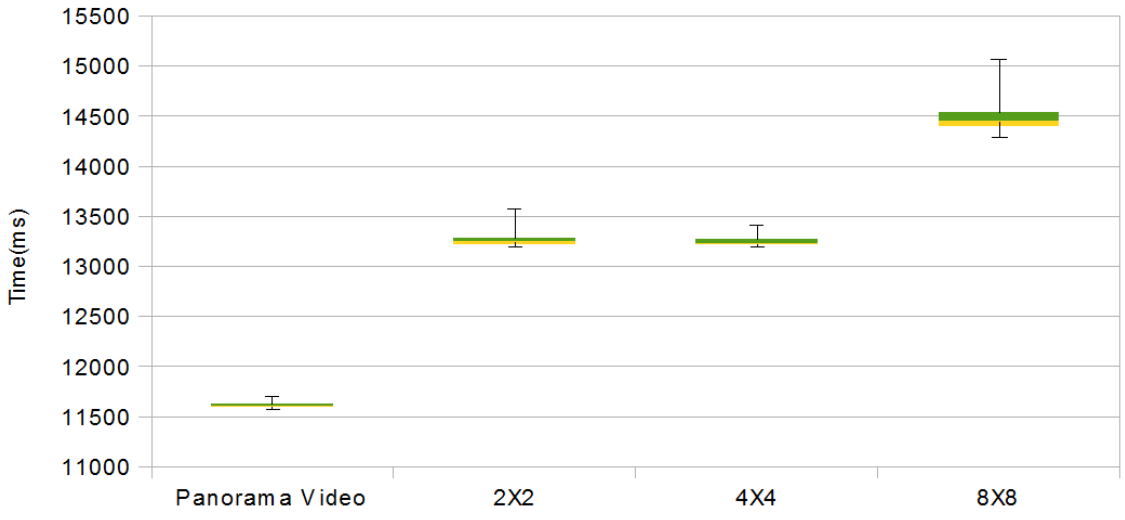



Figure 4.1: The time to partition and encode the tiles of the full panorama into multiple qualities with sequential encoding. These results originated from sequential encoding without the use of FFmpeg optimization

If we look at figure 4.1, the encoding latency between 2X2 and 4X 4 tiles does not seem to be any different. The upper bound of 4 tiles does seem to be a bit higher than the other, but it does not seem to introduce any new overhead by going from 2X2 to 4X4 tiling approach. From the results we can clearly see a big difference between the two previous approaches and 8X8. The dissimilarity is probably caused by the sheer amount of tasks the encoder has to complete. The total unique video streams we would have in 8X8 tiling approach is 64 tiles * 5 different qualities = 320 streams. Each video would need an AVCodecContext to describe how and in which quality it should be encoded in, and to use them we have to initialize the encoder with the context. The initialization and de-initialization of an encoder produces overhead thus prolonging the encoding time as shown in the same figure we mentioned before. But the main issue is in sequential processing, each operation is executed in order, thus we cannot proceed to the next tile and process it before the current tile has been completed. Encoding of a Panorama video into 5 different qualities had the least encoding latency of them all. That is because the overhead generated by tiling, such as calculating the offsets and the specific bytes we need to transfer from one place to another, is avoided. Additionally, encoding of panorama video does not suffer the same amount of overhead(creation and termination of encoders) as tiling.

**CPU and memory usage**

| *Panorama* | | *2X2* | | *4X4* | | *8X8* | |
|---|---|---|---|---|---|---|---|
| *CPU* | *RAM* | *CPU* | *RAM* | *CPU* | *RAM* | *CPU* | *RAM* |
| 99,25% | 18,73% | 98,58% | 16,99% | 98% | 16,49% | 95,42% | 16,39% |

Table 4.1: Sequential Encoding: Average CPU and Memory Usage

It was in our expectation that the results from using the sequential approach would not use the CPU efficiently. The utilization around 100% means that there was only 1 CPU core that did the encoding during the whole process. The memory usage seems to be normal, since an image from the panorama video would need to allocate more data as it is bigger. Thus the memory usage decreases the smaller a tile is. The decrease CPU usage is another aspect to look at. During profiling we noticed it had to make a lot of I/O, which is caused by the fact that each time a frame has been encoded, we will directly write it to file, thus the calls. As such the CPU will need to stop encoding and write the encoded bitstream to the file before it can proceed to the next frame. We can conclude that the first experiment is not adjusted to handle a huge amount of task at one.

### 4.2.2 Experiment 2: Sequential encoding of panorama video and tiles into multiple qualities with FFmpeg optimization

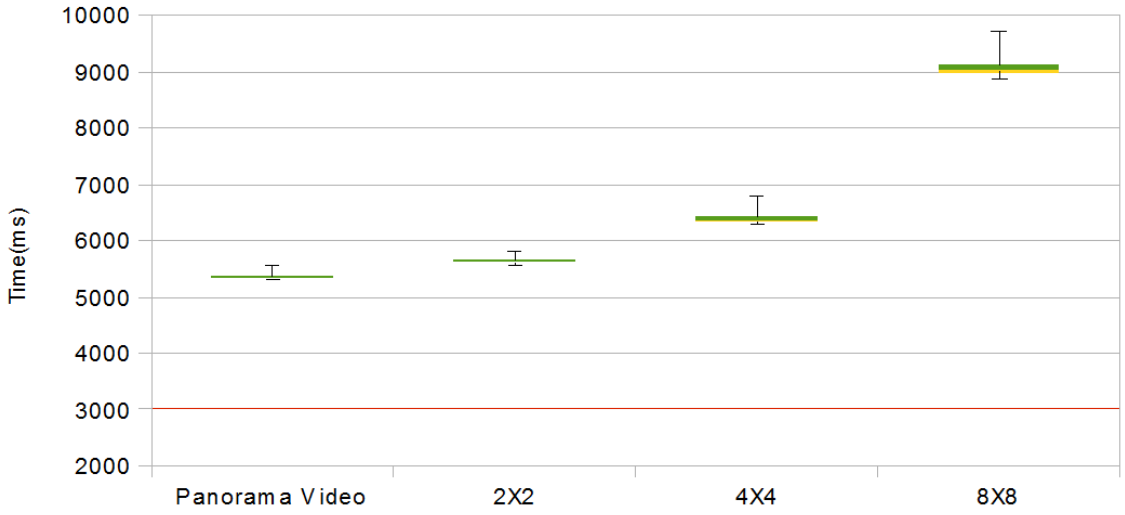



Figure 4.2: The time to partition and encode the tiles of the full panorama into multiple qualities with sequential encoding. These results originated from sequential encoding with FFmpeg optimization

We can see from figure 4.2 there was a considerable increase in encoding time using FFmpeg optimization compared to the previous test. A closer look at the code segment which controls the use of thread[14] (starting from line 161). FFmpeg would either initialize a number of thread corresponding to a given number by the user, or it will find out the the number of cores on the current machine and spawn the same number of threads. The threads would be put in a waiting state. Then whenever a frame is transferred to the encoder for coding, the frame will be split into slices and added to task queue. As soon as the queue is filled with the slices of the frame, a signal will be broadcast to every child thread which will proceed to encode each part of the frame. When encoding of a frame has been completed and there is no more task they will sleep again. Thus the overhead that comes with termination of threads can be avoided. It is a very efficient method for encoding, but there is a flaw. The optimization is designed to only encode 1 video stream as a time, as such they will be terminated when an encoder is longer needed. With the assumption that when an encoder is created with optimization on, it will create the same amount of threads as there are CPU cores thus we will get a formula:

$$encoders = tiles * qualities total\_threads = encoders * number\_of\_cores \tag{4.1}$$

With the equation 4.1, we can count the number of threads created and terminated during the whole encoding process as shown in table 4.2. The amount of overhead produced is closely related to the amount of tiles. Thus the encoding latency increases in parallel with the number of tiles, which can also be observed in figure 4.2.

#### CPU and memory usage

| *Panorama Video* | | *2X2* | | *4X4* | | *8X8* | |
|---|---|---|---|---|---|---|---|
| *CPU* | *RAM* | *CPU* | *RAM* | *CPU* | *RAM* | *CPU* | *RAM* |
| 577,5% | 24,05% | 480% | 18,46% | 439,2% | 16,94% | 300,89% | 16,53% |

Table 4.3: Sequential Encoding(FFmpeg Optimized): Average CPU and Memory Usage

Table 4.2: Overview of the number of total threads created and terminated after the whole encoding process has been completed using FFmpeg with optimization

| | Tiles | Qualities | Number of encoders | Total Threads created and terminated |
|---|---|---|---|---|
| Panorama Video | 1 | 5 | 5 | 40 |
| 2X2 Tiles | 4 | 5 | 20 | 160 |
| 4X4 Tiles | 16 | 5 | 80 | 640 |
| 8X8 Tiles | 64 | 5 | 320 | 2560 |

The utilization of the CPU was much higher than the previous experiment, but it does not seem like it can use more than 6 cores when encoding a whole panorama stream. The number of cores decreases the more tiles we have to encode. We assumed FFmpeg would spawn the same amount of threads as cores on the current machine, but it does not seem to be the case. Nevertheless with 3 threads per encoder when encoding a panorama video to 8X8 tiles, we would still get a huge amount of overhead.

### 4.2.3 Summary of Sequential Encoding

The performance of experiment 1 was poor due to its inability to use more than 1 core for encoding. With FFmpeg optimization, the utilization of CPU increased several times and encoding latency improved by a large margin. But it was not designed to handle a huge amount of video streams in one encoding session. Thus it can not be used in our tiling module, as such we can conclude that none of the sequential encoding approach would comply with our requirements.

## 4.3 Paralell Encoding

Parallel encoding stems from the idea of making use of the multi-core processors efficiently, which is the standard architecture of every CPU nowadays.

### 4.3.1 Experiment 3: Parallel Encoding using the amount of tiles and qualities to decide the number of threads

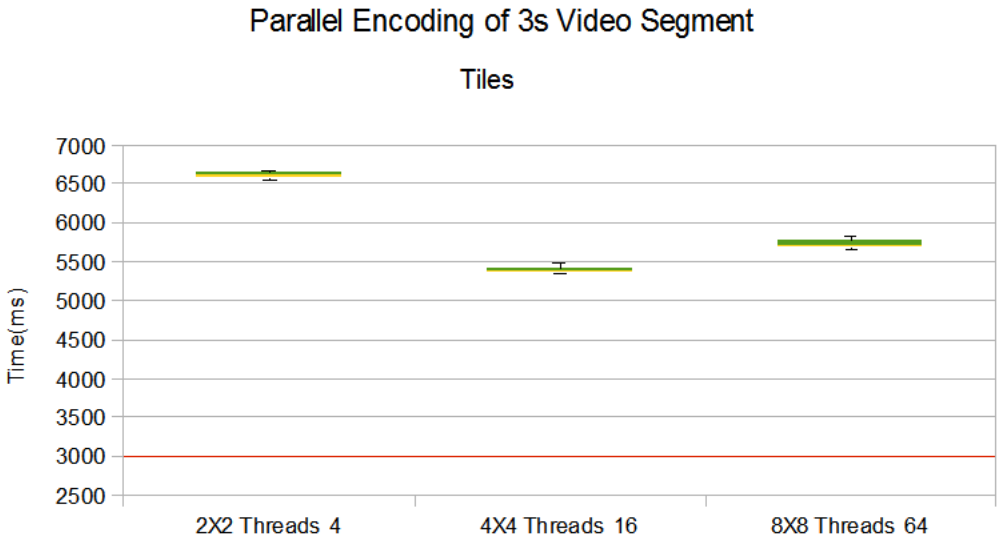



Figure 4.3: The time to partition and encode the tiles of the full panorama into multiple qualities with parallel encoding.We attained these results by using second concept, which is to spawn new threads for each tiles only.

We did not include testing of panorama video since we would only get the same results as sequential encoding by using 1 thread on it. We could use more threads for testing but it would only overlap other experiments.

As we can see from figure 4.3, encoding 2X2 tiles had the worst performance, this is due to the fact we had to terminate and create new threads each time a tile has been encoded and written to file. Another factor is that the tasks are much larger compared to 4X4 and 8X8. The performance of 4X4 with 16 threads came out best of the three test, but we have to keep in mind that the total amount of small tasks are considerable in 8X8. According to the results it seems that the there is a correlation in encoding latency and number of tasks. Even though it is not prominent, it is most like due to overhead produced when initialization and termination of the encoders.

#### CPU and memory usage

| *2X2* | | *4X4* | | *8X8* | |
|---|---|---|---|---|---|
| 4 Threads | | 16 Threads | | 64 Threads | |
| *CPU* | *RAM* | *CPU* | *RAM* | *CPU* | *RAM* |
| 352,22% | 19,75% | 718,06% | 19,94% | 736,46% | 18,51% |

Table 4.4: Parallel Encoding (Tiles): Average CPU and Memory Usage

The utilization of CPU was considerable higher than both sequential tests. There is also an increase of memory usage as a consequence of having more encoders working concurrently.

### 4.3.2 Experiment 4: Parallel Encoding using the amount of tiles to decide the number of threads

Each tile gets a thread, another experiments each stream gets one thread(Tiles*resolution)

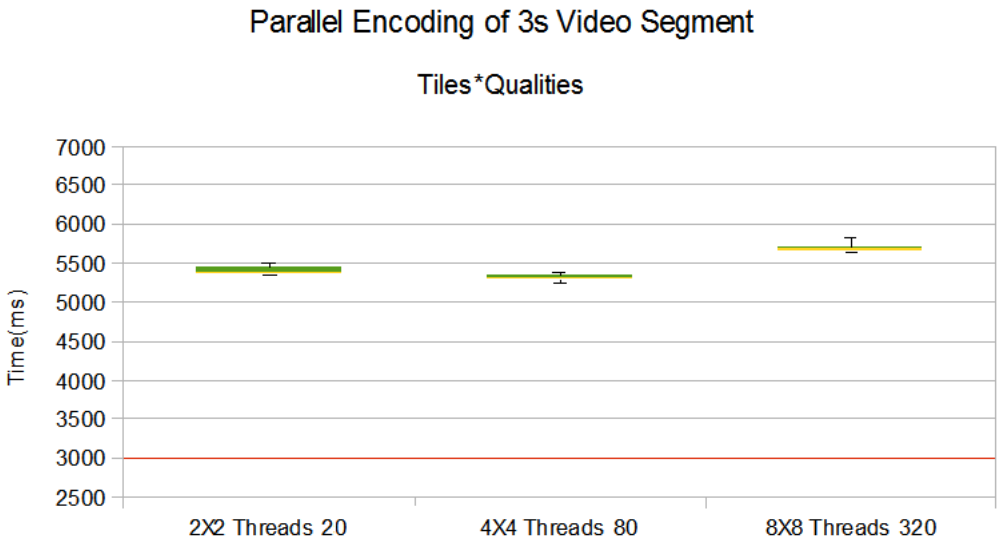



Figure 4.4: The time to partition and encode the tiles of the full panorama into multiple qualities with parallel encoding. We attained these results by using first concept, spawning new threads for each tiles and resolution

The results from 4X4 and 8X8 are almost identical to the previous test, but if we observe a little closer at the 8X8 results from both figures. The upper quartile of using 64 threads seems to be more concentrated(green line), which means there are more cases where it performed worse than normal. According to the all results till now, it seems context switching between threads does not cause much overhead compared to terminations of threads. Dedicating 20 threads to encode 2X2 tiles to 5 different qualities yielded much better results than the previous approach, which would indicate using more threads are more useful in our case.

**CPU and memory usage**

| *2X2* | | *4X4* | | *8X8* | |
|---|---|---|---|---|---|
| 20 Threads | | 80 Threads | | 320 Threads | |
| *CPU* | *RAM* | *CPU* | *RAM* | *CPU* | *RAM* |
| 730,6% | 30,45% | 762,18% | 21,06% | 762,21% | 18,72% |

Table 4.5: Parallel Encoding (Tiles * Qualities): Average CPU and Memory Usage

The memory utilization of 2X2 tiles approach were substantially higher than 4X4 and 8X8. This is caused by a task being larger and needs more processing time.The resolutions of 2X2 tiles are 2048x840, thus it will allocates more memory to store frame data for encoding. This behaviour can also be observed when transcoding a panorama video stream in experiment 2 at 4.2.2 using x264s frame slicing.

### 4.3.3 Summary of Parallel Encoding

Thread context switching usually produce a relatively high overhead. However, it did not affect our experiments using the parallel approach as much we would expect. Thus it seems there are situations where dedicating more resources are beneficial in as seen in 4.3.2. In our case, the performance boost is probably due to FFmpeg. As we know, we are using the libraries found in FFmpeg for encoding, and it has its own synchronization mechanism as mentioned in 3.8.1. Thus preventing collisions and shared data being overwritten between threads. When a thread calls a function from libavcodec, it will fetch a mutex, preventing other threads from

accessing it. This is probably the case between the two 2X2 tiling approaches. When using 20 threads, there would be a long queue waiting for a mutex, but during this time there would be context switching between the threads. Thus other threads can begin pre-preprocessing frame data before encoding them, which can be observed by the high usage of RAM in 4.3.2. Another factor which can be included are the Input and Output(I/O) operations. We are accessing the disk for writing, and during this time there will also be waiting too. The time threads has to wait are closely related to the size of an image surface they have to process. Thus we can conclude in this case, that dedicating more threads are beneficial when encoding tiles with larger resolutions for better utilization of system resources.

## 4.4 Thread pool Encoding, Persistent Threads

The concept of using persistent threads and having them wait for tasks was the third concept we introduced, and is the approach we wanted as an addition to the design. Thus it will be extensively tested and evaluated to see if we have successfully covered the challenges we had stated. We will be testing each tiling approach(2X2, 4X4 and 8X8) with different numbers of threads.

### 4.4.1 Experiment 5: Encoding a Panorama Video using different amount of threads

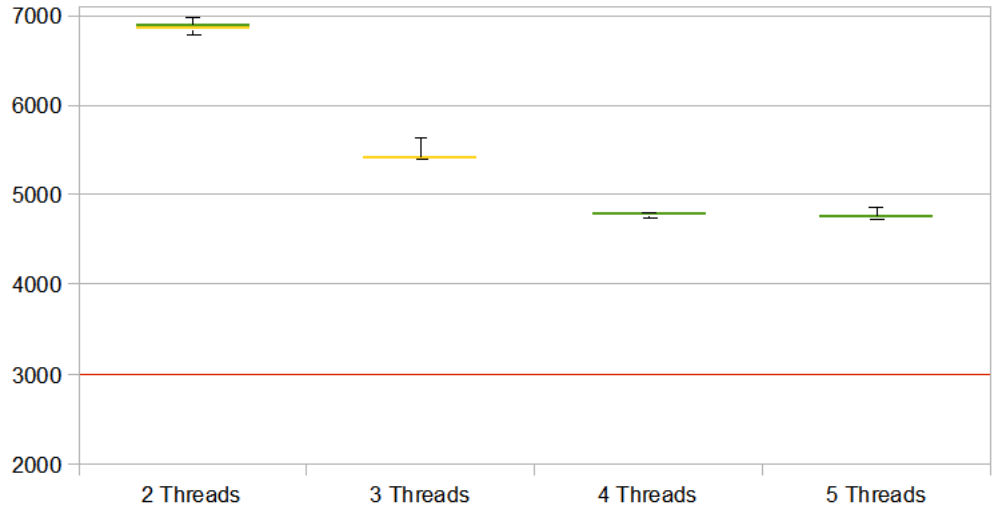



Figure 4.5: Thread-pool: Overview of the encoding latency of Panorama

Testing with 1 thread would deviate from the purpose of threading, thus ut was left out. According to the results in figure 4.5, there is a curve from 2 thread to 5 threads, with the lowest encoding latency at 5. There is a noticeable difference between using 4 and 5 threads for encoding. The same reason which caused the deviation between different number of threads from 4.3.3. where a thread has to wait for I/O operations and mutexes, can be applied here too.

Even though with overhead caused by waiting time, we can clearly see the increase in performance between using more threads to divide the workload, when compared to panorama video encoding found in 4.2.2 with frame slicing. Both approaches uses threads for encoding, but using persistent threads clearly had a positive affect on the performance.

Another factor is the tiling algorithm is not used on the panorama video, therefore the performance was considerably better than tiling.

**CPU and Memory Usage**

Table 4.6: Thread-pool: CPU and Memory usage during encoding of Panorama Video

| 2 Threads | | 3 Threads | | 4 Threads | | 5 Threads | |
|---|---|---|---|---|---|---|---|
| CPU | RAM | CPU | RAM | CPU | RAM | CPU | RAM |
| 182,5% | 21,78% | 290,33% | 24,47% | 356,75% | 26,3% | 499,83% | 31,6% |

We can notice there is a correlation between the CPU usage and the utilization of RAM in table 4.6. The results were within our expectation, since threads waiting for locks and IO operations would release CPU resource so other threads can acquire it and use it to pre-process their data. This could not be observed in the 3. and 4 experiments due to the different sizes of tiles each threads had to process.

### 4.4.2 Experiment 6: Encoding 2X2 tiles using different amount of threads

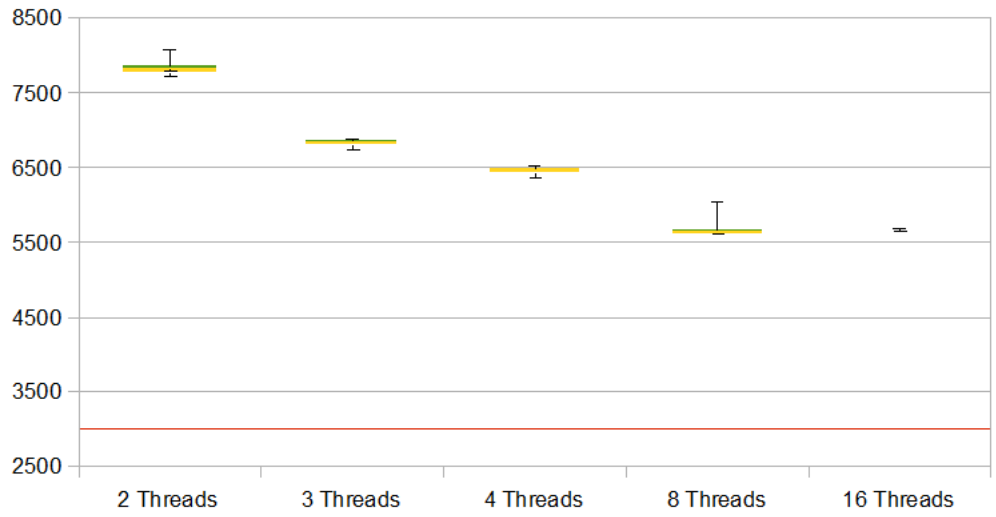



Figure 4.6: Thread-pool: Overview of the encoding latency of 2X2 Tiles

According to figure 4.6, we can observe a curve from using 2 threads to 16 threads for encoding, where using 8 threads had the least encoding latency. The performance between using 8 and 16 threads did not have a huge difference, but we can notice a slight increase in latency from using more threads. This could indicate the overhead caused by thread context switching. Dedicating 4 threads and under for encoding did not fare too well due to inefficient use of system resources and idling.

**CPU and memory usage**

Table 4.7: Thread-pool: CPU and Memory usage during encoding of 2X2 Tiles

| 2 Threads | | 3 Threads | | 4 Threads | | 8 Threads | | 16 Threads | |
|---|---|---|---|---|---|---|---|---|---|
| *CPU* | *RAM* | *CPU* | *RAM* | *CPU* | *RAM* | *CPU* | *RAM* | *CPU* | *RAM* |
| 174,78% | 17,44% | 258% | 17,74% | 313,33% | 17,62% | 470,6% | 18,48% | 454,5% | 18,85% |

In table 4.7, we can see the utilization of the CPU was slightly higher when using 8 threads than 16 threads. By comparing the results to figure 4.6, we can notice there is a correlation between CPU usage and encoding latency.

### 4.4.3 Experiment 7: Encoding 4X4 tiles using different amount of threads

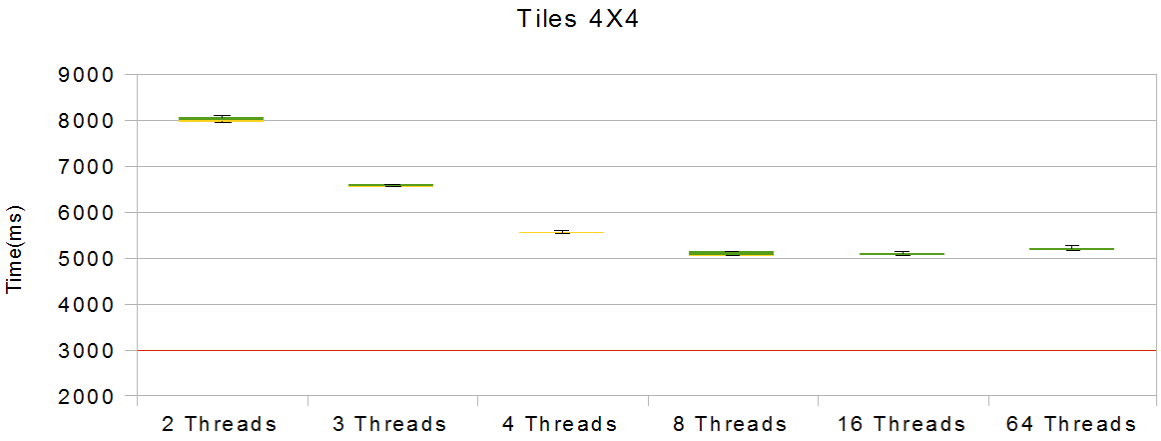



Figure 4.7: Thread-pool: Overview of the encoding latency of 4X4 Tiles

We could not use 64 threads in the previous experiment due to the fact 2X2 tiles did not have enough tasks to be divided between 21 threads and beyond (4 tiles * 5 qualities = 20 tasks/video streams). Thus we will not be able to make any comparisons on 64 threads against previous work.

Compared to the previous results from experiment 6, we can see from figure 4.7 there is some similarities and differences. Using 2 and 3 threads for encoding shared the same issue between them, but using 4 threads and beyond yielded a performance leap. The increase was around 600ms on the aforementioned instances. The reason for this is the tile sizes. As we know, video streams are made up of a sequence of images, and these frames are encoded subsequently. It is when we encode a picture where we utilize the FFmpeg library that we encounter mutexes, thus we have to wait. But since the resolution of these tiles has been reduced, the execution time to process each frame has also been reduced significantly. This also applies to IO operations. Still it had some affect on the result from using 4 threads.

#### CPU and Memory Usage

Table 4.8: Thread-pool: CPU and Memory usage during encoding of 4X4 Tiles

| 2 Threads | | 3 Threads | | 4 Threads | | 8 Threads | | 16 Threads | | 64 Threads | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| *CPU* | *RAM* | *CPU* | *RAM* | *CPU* | *RAM* | *CPU* | *RAM* | *CPU* | *RAM* | *CPU* | *RAM* |
| 194,86% | 16,9% | 287,33% | 17,17% | 373,8% | 17,46% | 786,25% | 18,38% | 789,25% | 20,65% | 775,25% | 21,3% |

When we compare the previous CPU and memory usage, there was a significant increase in utilization of system resources which can be seen in table 4.8. This verified our deduction, where the increase of the performance was caused by the reduction of waiting time produced by mutexes and IO operations.

### 4.4.4 Experiment 8: Encoding 8X8 tiles using different amount of threads

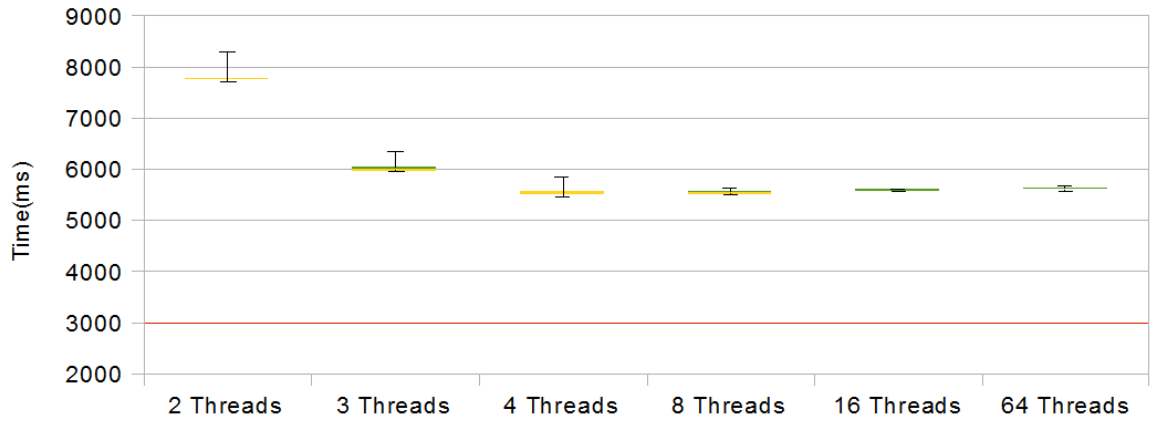



Figure 4.8: Thread-pool: Overview of the encoding latency of 8X8 Tiles

The results of experiment 8 can be seen in figure 4.8, displays a convex curve starting from the 2 threads to 64 threads with the lowest point at 4 threads. The increased encoding latency is a result of increasing the number of tiles, thus workload. But the latency which was caused by idling of threads has been reduced significantly, thus the difference between 4 threads and 8 threads was also affected. However, the best case is still using 4 threads for encoding of 8X8 tiles in 5 different qualities.

#### CPU and Memory Usage

Table 4.9: Thread-pool: CPU and Memory usage during encoding of 8X8 Tiles

| 2 Threads | | 3 Threads | | 4 Threads | | 8 Threads | | 16 Threads | | 64 Threads | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| *CPU* | *RAM* | *CPU* | *RAM* | *CPU* | *RAM* | *CPU* | *RAM* | *CPU* | *RAM* | *CPU* | *RAM* |
| 173,25% | 16,51% | 282,6% | 16,62% | 335,33% | 16,68% | 757,25% | 17,28% | 765,2% | 17,98% | 758,6% | 19,36% |

We can see from the usage of system resource in table 4.9 does not correlate with the results from figure 4.8. In this case, we are using a Intel CPU with 4 cores, thus we can deduce that by utilizing more than 4 threads, it would increase the overhead from context switching between threads. It can also be observed in the same aforementioned figure.

## 4.5 Storage consumption

Table 4.10: Overview of the storage consumption

| | |
|---|---|
| Source Panorama | 2.9M |
| Panorama Video | 7.0M |
| 2X2 Tiles | 12M |
| 4X4 Tiles | 13M |
| 8X8 Tiles | 15M |

The storage consumption of different tiling approaches can be observed in table 4.10. We can observe the space requirement increases in parallel with the number of tiles. There are two

reasons we can think of. In most cases, we need to create a stream header that contains the video files meta-data. They are necessary for playback devices to recognize the structure of a video stream so that they are able to play them. Thus the increased space consumption.

Another factor which is probably responsible for this behaviour is Constant Rate Factor. As we know from , CRF can vary how much quantization can be applied to each frame by exploiting the fact that the human eyes are better at observing still images than fast moving objects. Thus it can increase the Quantization Parameter(QP) on frames where there are moving objects, and decrease QP on still pictures. The panorama video we are using is a soccer game which has a duration of 3 seconds, and during that time there was little movement involved. If we observe figure 4.9, there are many places where there are either no activity or will never have at all. Hence, by separating the video to smaller pieces, we inexplicitly created more streams with no motion, which makes CRF increase its QP which increases quality and as a consequence storage consumption.

### 4.5.1 Reducing bandwidth requirement with the use of tiling

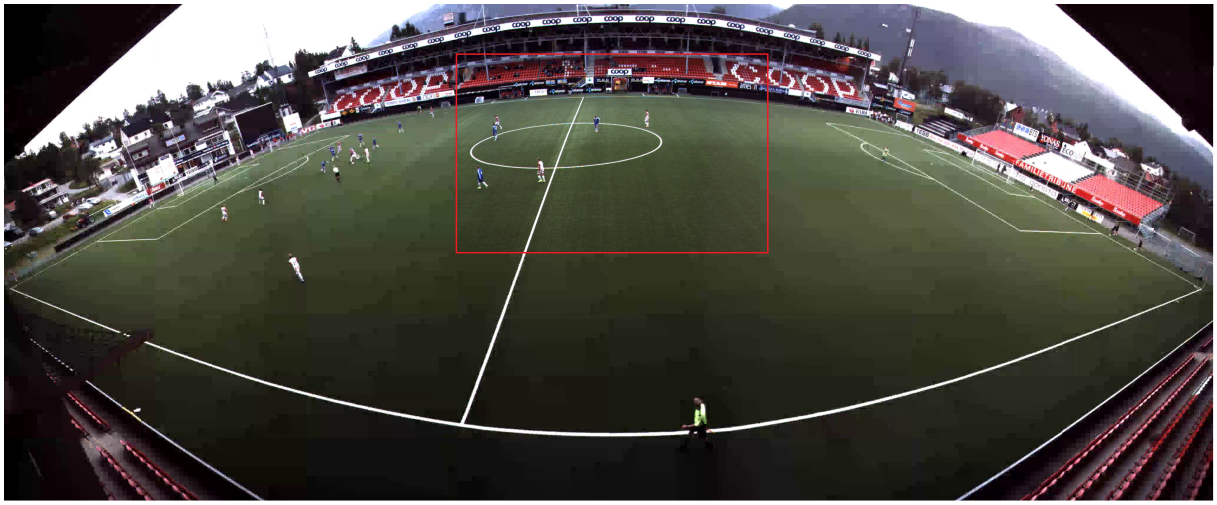


Figure 4.9: Panorama video mixed with tiles from 8X8 with CRF 48, 1,06MB

Figure 4.10: Source panorama video , 2,87MB

Though the storage consumption was increased due to tiling there should still be no problem since harddrives are inexpensive nowadays. But with the increase of storage, we can offer a significant reduction in bandwidth requirement which can be observed in the figures 4.9 and

4.10. As we mentioned before, interactive virtual cameras are used in many technologies today for streaming of high quality content. The cameras are able to zoom, pan and tilt which we can exploit using tiling. We can do that through sending low quality tiles on areas where the camera is not focused on. However, high quality tiles will be transferred to the part where the camera are concentrated on, which is also know as Region of Interest(RoI). The RoI can be observed in the first figure with the red mark which represents the camera.

### 4.5.2 Further reducing the storage consumption

We did initial test at the beginning by experimenting with different presets. The results was *ultrafast* preset yielded the best performance, but the preset *veryfast* was not far behind and the video compression was significantly more efficient. We wanted to redo the test after the final design has been completed, so that we can find out how much storage consumption we can potentially reduce.

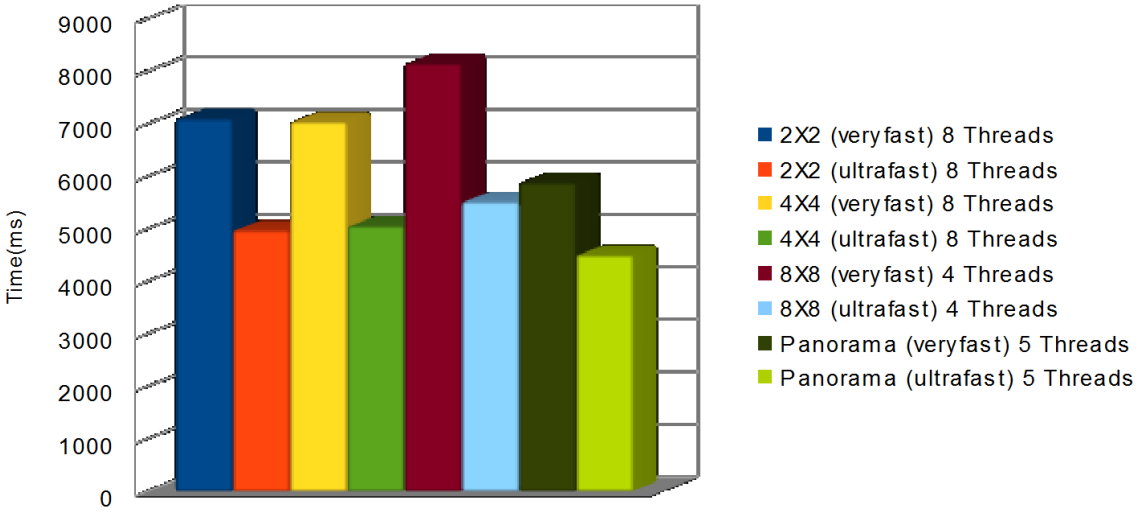



Figure 4.11: Encoding latency presets

The number of threads chosen for each tests in figure 4.11, were based on the best cases for each tiling approach. We can notice there were no tests where *veryfast* preset triumphed. It was in our expectation, but the difference in compression can be seen in table 4.11. In this thesis, encoding latency was the main focus thus we did not try to further reduce the storage requirement.

Table 4.11: Storage requirement

| | Panarama | 2X2 | 4X4 | 8X8 |
|---|---|---|---|---|
| CPU ultrafast | 7.0M | 12M | 13MB | 15MB |
| CPU veryfast | 1.2M | 4.8M | 5.1M | 6.0M |

## 4.6 Summary

We have presented multiple tests where we confirmed the liability of sequential encoding, and verified to ourselves to come up with new approaches and ideas to further improvise our design and implementation. Even though using frame slicing from FFmpeg improved sequential encoding performance drastically, it was not compatible with our approach with tiling due to the fact that we required a huge amount of encoders which can be executed

simultaneously. The overhead from frame slicing was produced by the creation and termination of threads, which materializes whenever a codec is initialized and terminated. Thus the performance decreased whenever the number of tiles increased. From experimenting with parallel encoding of tiles, we discovered the efficiency of threads but also the overhead caused by thread context switching. It also gave us an indication of how much overhead was caused by IO operations and waiting for mutexes. Through the experiments, we designed the concept of using persistent threads to avoid unnecessary overhead by terminations of threads during an encoding session. After extensive testing with the last concept we can conclude that in video encoding, being CPU-bounded, does not always yield the best in every situations. Synchronization mechanism and IO operations plays an important factor, which can be observed during encoding of 2X2 and 4X4 tiles. With larger tiles, the duration to process a frame increases and as a consequence other threads are prevented to access a specific resource. The same condition also applies to IO calls, larger tiles more data to write. Thus with 2X2 and 4X4 approaches, we can safely conclude that utilizing Intel Hyper-Technology would produce the best results, since resources are released when a thread is idle and other threads can claim it and pre-process their data. However, using 4 threads in 8X8 tiling approach generated the best performance. The extra overhead which was produced by IO operations and locking mechanism, had been reduced significantly due to tiles being smaller and as a consequence less processing time is needed to complete a task.

Even though the storage requirement for tiling is significant, we can exploit the limited vision of RoI and adapt the tiles in accordance based on the camera placement on video, thus reducing the bandwidth requirement. Methods for reducing the space required was also proposed, but the trade off is between encoding latency and compression efficiency.

# Chapter 5

# Hardware Encoding: Design and Implementation

/

## 5.1 Introduction

Hardware encoders are dedicated processors that has been around longer than general purpose processors, th In this chapter we will present a hardware encder which we will integrate it in our design, and test its performance. There were also three hardware encoders that was considered. It was Intel QuickSync, AMD VCE and NVENC. Intel QuickSync was considered because the test machine we are using has an Intel core with QuickSync. But it was not easy to activate and since it sits on the die, QuickSync is not scalable without any specialized hardware. AMD VCE should have been a good choose to use, but there were no available card in proximity. Thus we decided to use NVENC since it was the only one that was easily obtainable and scalable with our design and implementation.

## 5.2 NVENC

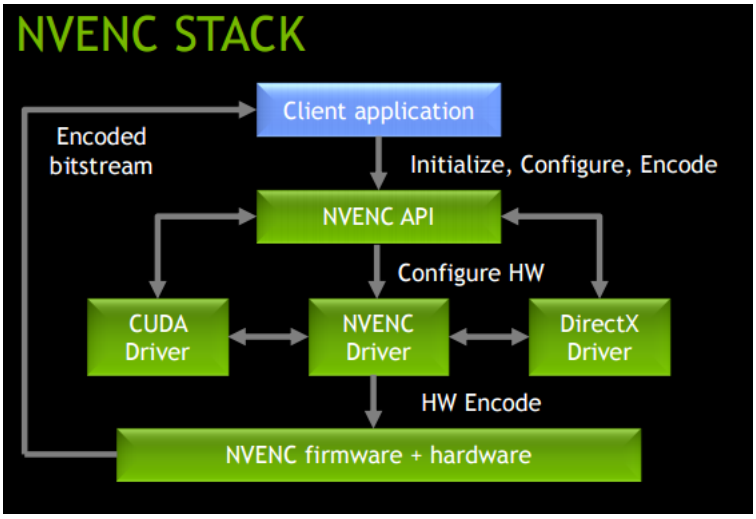



Figure 5.1: Detailed overview of NVENC encoder API[7]

A detailed overview of the NVENC stack can be observed in figure **??**. Variants of Maxwell architectures has 2 instances of NVENC encoders to further improve performance. Context switching between those encoder are done by the stack automatically.

#### NVENC SDK 3.0

In SDK 3.0 there is no support for asynchronous mode(will be mentioned later), but it allows us to utilize more than 2 encoding sessions simultanously.

#### NVENC SDK 5.0

Does not have asynchronous mode either, but the structure/framework ahsa been updated thus it performance better than SDK 3.0. The downside is in this version they removed the ability to add a licence to the nvencApi, thus adding a limit on how many concurrent encoding session we can have .

## 5.3 Pixel formats supported

NVIDIAs hardware encoder NVENC can only accept pixel format NV12 as input, as it is hardware based, we cannot solve it by updating or modifying the hardware. Thus the only solution was either use another panorama video as input with pixel format NV12, or we can use the libswscale library from FFmpeg to convert the pixel format to the corresponding format NVENC accepts. Followed with NVENC SDK 3.0 and 5.0.1 packages, there was a sample program which accepts YUV 4:2:0 as pixel format as input. We tested it with with YUV 4:2:2 too, but there seems to be some visual artifacts, which we will be explained in more detail in 5.6.1. Since we can potentially use pixel format YUV 4:2:0 as an input in the sample program, it will be experimented on and evaluated against NV12. The test will reveal which format is best suited for more comprehensive experimentation on the hardware encoder. Even though it accepts only NV12, the encoded video output streams are recognized as in pixel format YUV 4:2:0 by FFmpeg.

### 5.3.1 YUV420

YUV is a color space which can represents most colors a an can perceive and it is commonly used in video applications. More detailed explanation about YUV has been discussed in 3.6.1. The difference is how much sampling has been done on a particular version of YUV. The numbering found after the initials YUV describes sampling rate. For example in YUV 4:4:4, all the Y, U and V channels has the sme sample rate, and in YUV 4:2:2 U and V is sampled half the rate of Y. For YUV 4:2:0 the sampling rate is also half of the Y rate, but both the vertical and the horizontal directions. Thus it reduces the bandwidth by 50%. The YUV 4:2:0 planes are illustrated in figure 5.2.

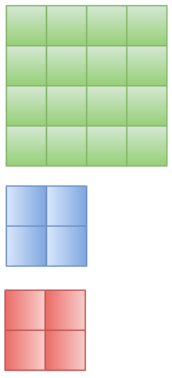


Figure 5.2: YUV 4:2:0

### 5.3.2 NV12

As mentioned before, NV12 is the only pixel format NVENC accepts. Thus conversion is necessary if other pixel formats are to be used. In NV12 the chroma components are also sampled half the rate of Y channel both vertically and horizontally. But the U and V channels are interleaved where every other value is either U or V, thus NV12 has only 2 planes. Illustration of NV12 with 2 planes can be seen in fiugre 5.3

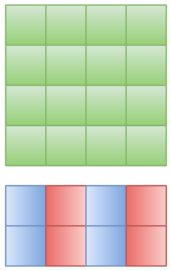


Figure 5.3: NV12

## 5.4 FFmpeg with NVENC support

At the initial phase of our design, we discovered that FFmpeg supported the use of NVIDIA hardware encoder. To be more precise, we found a repository[15] which implemented the support for using NVENC through FFmpegs generic functions and structures. Therefore we decided to test it to have an indication of how the performance is, and verify if it is plausible to use it in our design and implementation. The tutorial of how to build and configure FFmpeg to support NVENC is found in the same repository, thus we would not go into detail of how to do it.

### 5.4.1 Testing FFmpeg with NVENC support

The process for encoding a video stream is very similar to our implementation of the software encoder. It can be rounded up with these simple steps:

1. Open stream
2. Read video file
3. Decode it and store every frames extracted from file
4. Populate the necessary information in an encoder context
5. Fetch an encoder from library corresponding to format you want
6. Encode all the frames and store it in disk

As we can see from these steps, the procedure to use NVENC through FFmpeg is very similar to our implementation of the software encoder. But instead of fetching libx264 from the library, we will ask for libnvenc. Libnvenc like libx264, is a wrapper or implementation that are necessary to push video data from the FFmpeg internal format to an external encoder, in this case the hardware encoder NVENC.

We ran some tests on it and the results were comparable to the sequential encoding with optimization from FFmpeg. When we tried to integrate the concept of parallel computing to our implementation we ran into some problems. The first problem we faced was when we used two threads for encoding, the latency increased exponentially. We knew there was some overhead when encoding is executed simultaneously, but the performance has never been so poor. Even though we are unsure what provoked this kind of behaviour, we decided to

apply more threads thus encoders to find out if there are any correlation between the number of threads and the latency. It was then we encountered another obstacle. When we tried to initialize encoding session beyond the second one it failed. We were unsure whether it was a bug in the implementation or some settings that was necessary to set but overlooked by us. Thus we decided that it was essential to gain more knowledge of how NVENC works so we proceeded use the sample program followed with NVENC SDK 5.0.1 package. During experimenting and testing with NVENC API, we found out from the NVENC SDK homepage[37] and PDF documentation[38], that our current NVENC SDK package(version 5.0) only allows up to two simultaneous encode sessions per system for low-end Quadro and GeForce GPUs.

### 5.4.2 Downgrading drivers to open more encoding sessions

According to one of our colleagues, it was necessary to have a special licence key to use NVENC to be able to initialize more than 2 encoding session. But during testing of the sample program, we did not provide a licence key. It looks like NVIDIA removed the requirement of a license key to use NVENC and it also limited the number of concurrent encoding session through GPU driver updates. With a closer look in the Application Program Interface(API) from NVENC 2.0, 3.0, 4.0 and 5.0. It seems like they removed NV_ENC_OPEN_ENCODE_SESSION_EX_PARAMS:: clientPtr, which was used to set up an encoder and pass the key for verification, from 4.0 and beyond. Thus in order to use the licence, which we obtained from our colleague, we needed to use an older API from NVENC 3.0. The newer drivers are not compatible with version 3.0 thus we had to downgrade NVIDIA driver to version 334.21. After we have configured our machine, we ran some initial tests and verified that it could open more than 2 encoding sessions simultaneously.

## 5.5 NVENC initialization

We used the libraries from FFmpeg to read and decode H.264 files and store their raw video streams as frames in memory in the software encoder implementation. Instead of making a new implementation with the same procedure of opening, reading, decoding and storing the necessary data, we decided to try and integrate the sample program from NVENC SDK 5.0.1 to our software encoder implementation. Even though we had several problems with the integration caused by the difference of structures, function calls and parameters found in the API from 3.0. The process would still be easier than integrating the sample program which followed the NVENC SDK 3.0 package to our program. First of all, it was designed to be a standalone package with no need for external libraries. It contained a huge amount of libraries which was custom made by NVIDIA, such as thread handling, synchronization libraries and many more. Debugging and trying to optimize the implementation proved to be too difficult, thus the decision we made. In NVENC SDK 5.0.1, NVIDIA removed almost all of their custom libraries and there was only the API left. They still had their framework of containers which does some pre-processing of frames or presets before encoding, but it does seem to support threading. The containers are created as objects and the functions are all private thus making external access impossible. The difference in API was not the only challenge we had to overcome. There was a lot of redundant data and even though at first glans it seems to support threading, it performed horrible when tested it. Thus there was many improvements that we made during the design and implementation of our program.

### 5.5.1 Optimization and reducing unnecessary overhead

During testing of our new implementation, which supported NVENC, we discovered that the program produced to much overhead and sometimes it even crashed. In Ubuntu(or Linux enviroment), we can use nvidia-smi -l 1(comes pre-installed when using NVIDIA drivers), to

see some information of what is happening during execution. The operator -l means that the program will loop or probe the GPU in a given interval(in our case 1 second) to get the current status of it. What we noticed was, whenever the memory usage exceeded the video RAM it will cause a crash. Which is normal, but the video files we are processing should not have that much impact on the memory usage. Maybe close to the limit but it should not exceed. So we tried to increase the number of threads one by one until we hit the threshold. It turned out be the initialization of a CUDA context. Each time a context is initialized and allocated, it will require a huge amount of memory, hence appointing more threads would only create more CUDA context which leads to our crash. After some testing with encoding video streams in different tile formations, we found out that it was possible to allocate one CUDA context that could be shared between several session encoders. How many encoders that can share a context will vary depending on the resolution of a video or tile. Also CUDA context switching is expensive which could lead to performance loss.

### 5.5.2 Initializtation of CUDA context

As mentioned in 5.5.1, instead of creating a CUDA context of each thread, we will only create it once and make it shareable to avoid crash and loss in performance. A CUDA context can be described as a CPU process. Before a context switching in CPU, general registers, segments registers, Floating-Point Unit(FPU) registers, stack pointers, allocations and other state-related information associated with a process will be stored before switching to another process. For GPUs, all the state related information of a particular process is stored in a CUDA context. Creating a CUDA context can be done with these steps:

1. Initialize the driver API with cuInit()

2. Get a handle or pointer to the device(GPU) with cuDeviceGet()

3. Create a CUDA Context associated with the callee through cuCtxCreate()

4. To make it shareable we used cuCtxPopCurrent() to release the context from the host thread, so other threads can access it

### 5.5.3 NVENC Encoding configurations

NVENC has many different options to configure much like x264, and it has also predefined configurations which controls various encoding parameters for motion estimation(ME)and which rate control scheme to use in H.264 encoder. Thus it can be used in many different scenarios if the user do not want to manually change or set each encoding parameters. The following presets which are supported in NVENC SDK 3.0 can be seen in figure 5.4.

[0]refer to ME

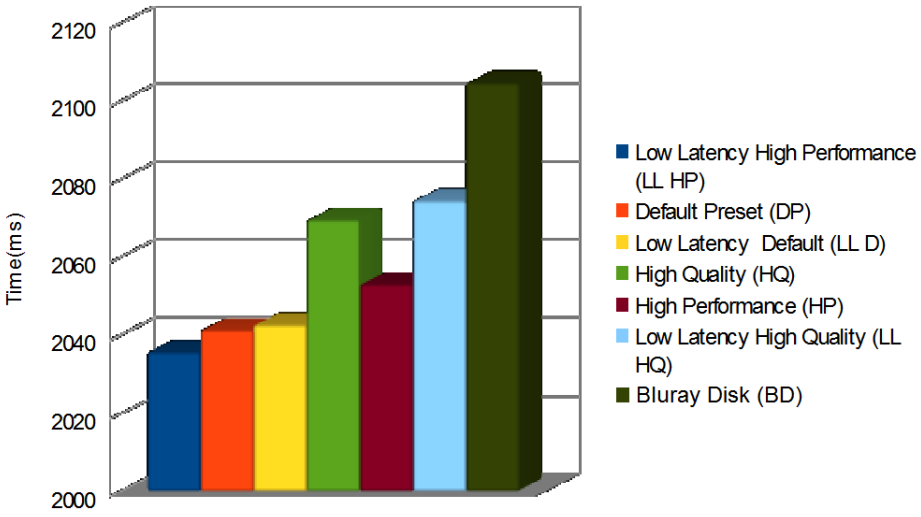



Figure 5.4: NVENC: Testing different presets configuration with encoding a Panorama Video to 1 quality

Even though the name of the presets were pretty self-explanatory, there were many of them that focused on low latency. So we ran some test to see if there are any large difference between the presets. From figure 5.4, we can see some irregularities, since we expected HP to be faster than HQ. But it was probably affected by our own settings that we made which overrides configurations made by the presets. The preset we are mainly interested in are the one which gives us the lowest latency, which is Low Latency High Performance. This preset are designed for speed, as such it will disable the use of B-frames avoiding the use of more advanced encoding algorithms for video compression. The configurations we manually changed was:

1. The size of Group of Pictures(GoP)

2. The rate of frames per seconds (FPS)

3. The height and width

4. Number of B-frames

FPS decides how many frames are shown during playback, the size of GoP is mentioned in 3.5.2, the height and width is set depending on whether the output is a panorama video or a tile. With a tile, the resolution is set based on which tiling approach we used. The next parameter tells NVENC how many B-frames it will use during encoding. NVENC SDK 3.0 is bundled with different codecs capable of handling many formats such as Joint Photographic Group(JPEG)[1], Windows Media Codec(VC-1)[2], MPEG-2 and some more. Thus we have explicitly set the codec to H.264 for NVENC to encode the data streams according to H.264 standard. Other formats are not our main focus in this thesis hence we will not discuss about them further. NVENC has many rate control modes that can be used to dynamically adjusts the encoder parameters to achieve a specific bit rate. The ones NVENC supports are as follows:

[1]A method of lossy compression of digital images
[2]Video codec developed by Windows

| |
|---|
| **Constant Quantization Parameter mode (CONSQP)** |
| **Variable bitrate mode (VBR)** |
| **Constant bitrate mode (CBR)** |
| **Variable bitrate mode with MinQP (VBR_MINQP)** |
| **2-pass encoding optimized for image quality and works only with low latency mode (2_PASS_QUALITY)** |
| **2-pass encoding optimized for maintaining frame size and works only with low latency mode (2_PASS_FRAMESIZE_CAP)** |
| **Constant bitrate mode using two pass for IDR frame only (CBR2)** |

Table 5.1: Rate control modes supported in NVENC

There does not appear to be a support for Constant Rate Factor in NVENC which we used in x264. It does seem like two of the modes in NVENC supports the use of QP, CONSQP and VBR with minumum QP. As we are trying to set the encoding parameters to conform to the options we used in x264, only these two are relevant. We tested these modes and it seems CONSQP was faster, but VBR had lower storage consumption. Thus we are going to use CONSQP as it gives us the lowest latency. The reason for the sprecific choices with the configurations we have in NVENC is to have a more evaluation between the different encoders when we compare them.

### 5.5.4 Performance bottleneck?

Another configuration is automatically set based on which Operative System(OS) enviroment you are on, and that is synchronous and asynchronous modes. The difference between them is:

**Synchronous** Encoding of frames can only happen in sequential order, a new frame cannot be encoded before the previous frame has been completed

**Asynchronous** Multiple frames can be encoded simultaneously

The advantage of using the first mode is we do not need to keep track of the order of the frames, encoded frames we get from NVENC is the same one we fed it. In asynchronous mode we can encode multiple frames simultaneously, but we have to keep track of each of the frames so we can write them in the right order. Both of these modes are supported for Windows, but in Linux the only mode that is supported is synchronous mode. These restrictions applies to every NVIDIA Video Interface versions(NVENC SDK 1.0, 2.0, 3.0, 4.0, 5.0) that is currently available.

### 5.5.5 Allocation of Input and Output buffers for encoding

An encoder with the configuration we made in 5.5.3 can be created through the API provided by NVIDIA. Before we can use it, we have to allocate input and output buffers which will be used as a handle by NVENC. The buffers are used to store the input frame data which will be transferred from system memory to video memory for encoding, and when it has completed the process then the encoded frame will be transferred back and stored in the output buffer in system memory. We can allocate NVENC input buffers with NvEncCreateInputBuffer and output buffers with NvEncCreateBitstreamBuffer.

### 5.5.6 Optimization to the intergration of hardware encoder

At the initial phase of integrating NVENC to our implementation, we followed the same encoding procedure we had in the software encoder design. We will create an encoder, use it to encode a video file and then terminate it. Thus the general flow of the program was:

1. For each tasks do the following
    (a) Create an encoder
    (b) Encode the video file
    (c) Terminate the encoder

This design did not fit well, as it caused huge overhead most likely due to creation and termination of the encoder. The same problem did not affect as much to software encoder because FFmpeg is constantly developed and updated, as such it is highly optimized. We found out later that in NVENC SDK, there was a function NvEncReconfigureEncoder which was designed to configure an encoder during an encoding process. So we implemented it into our system thus avoiding the creation of a new encoder and termination of it for each Task. So the general flow has been updated to:

1. Create an encoder
2. For each task do the following
    (a) Reconfigure the encoder corresponding to task
    (b) Encode the video file
3. Terminate when there are no more task

## 5.6 Encoding process

The procedure of pre-processing of a frame prior encoding is very similar to the procedure in found in the implementation of software encoder. Before proceeding to encode a video file and write it to file, we will reconfigure the encoder to correspond to the parameters found in the task we got from getTask().

### 5.6.1 Conversion of a frame

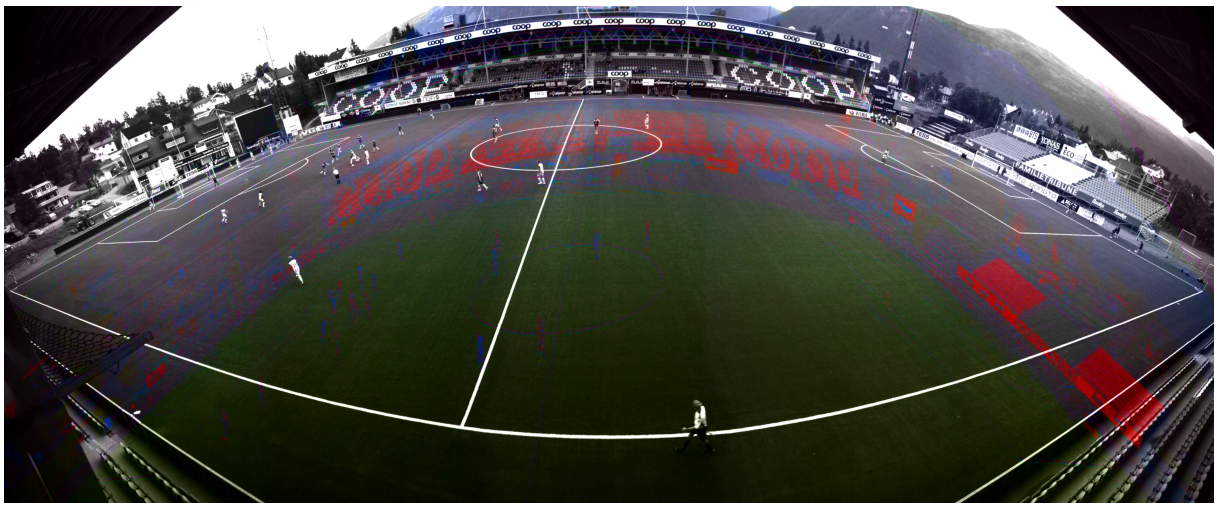


Figure 5.5: NVENC YUV422 wrong calculations

The panorama video files used in this thesis and implementation is in pixel format YUV422 as mentioned before. Using the sample code we managed to encode video files with YUV422 as pixel format. But as you can see from figure 5.5, it did not seem to make the right calculations. However, with YUV420 as pixel format it came out without having any visual artifacts. The only color space NVENC accepts is NV12, thus we have to convert YUV422 to either NV12 or

YUV420(with the conversion function found in the sample code). Among the libraries found in FFmpeg, libswscale is the library which contains highly optimized color space and pixel conversion and image scaling operations.

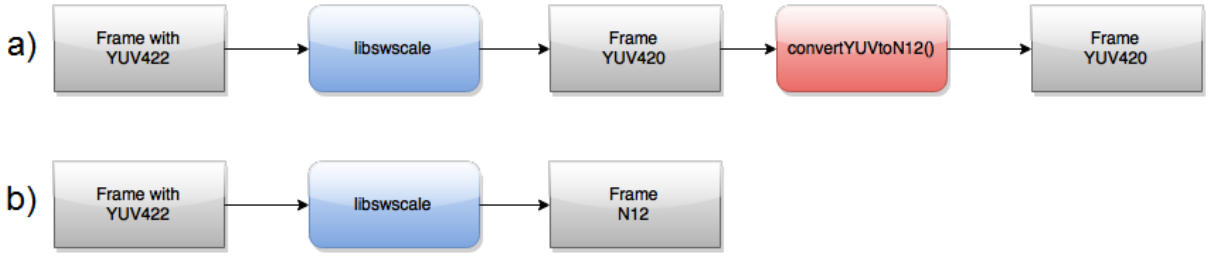



Figure 5.6: Conversion steps

Usually it would be logical to take the shortest conversion path as shown in figure 5.6, but when we tested the latency between those two route we got some very intriguing results.

| What format to convert first | YUV420 | NV12 |
|---|---|---|
| FFmpeg libswscale | 468.45ms | 911.53ms |
| convertYUVtoNV12 | 227.20ms | 0.00ms |
| Total time used for conversion | 695,65ms | 911.53ms |
| Total time of encoding | 1669.80ms | 1948.86ms |

Table 5.2: Overview of overhead produced by conversion

The results in table 5.2 is gained from encoding a full panorama video into 1 high quality. If we follow the a) route in figure 5.6, there will be an overhead caused by the extra conversion found in the sample code. With the b) route, we avoid the overhead. Still the additional step we took, where we convert YUV422 to YUV420 using libswscale then proceed using convertYUVtoNV12 for conversion to pixel format NV12, had the lowest encoding latency. Between the two route, a) was 48,61% faster than b). The reason at the moment is unclear, but we will use the YUV420 as the pixel format to convert to NV12 since it gives us the least encoding latency.

### 5.6.2 NVENC resolution bug

When we encoded a panorama video into 64 tiles(8X8), the resolution of each tile was 512x210. There seems to be a issue of the resolution because we get a green line at the bottom as seen on figure 5.7. In the previous implementation using libx264 for encoding, the same resolutions did not appear to be a problem. Further reading revealed that most video codec nowadays split a frame or an image for compression into smaller chunks called macroblocks, rather than processing the whole image at once. These blocks are usually fixed in size, such as 4x4, 8x8, 16x16 pixels thus the resolutions has to be a multiple of 4, 8 or 16 depending on the block size. Truncating the height to 212 and 214 did not help, but with 216 we got every pixels we needed. Thus based on our observation and the fact that every other resolutions we used in the various tiles formations we have, the size of a macroblock used in NVENC has to be 8 since 216 is not a multiple of neither 4 nor 16.

Figure 5.7: 8X8 tiling approach, resolution 512x210. Missing some rows of pixels

### 5.6.3 Encoding of a frame - Encoding module

Before we can encode a frame, it has to pre-processed the same way as the procedure found in . Then we will fetch an input buffer which we allocated beforehand and lock it. To write data to input buffer, we will have to convert frame data format to NV12 using convertYUVtoNV12() since we decided to use pixel format YUV420 instead of NV12 to decrease the latency. We will not go into detail of how it works, but basically convertYUVtoNV12() will align the pixels from YUV420 to conform the NV12 pixel format and write it to the input buffer. When it has been converted and completed the writing to the input buffer, we will need to unlock it. Before we encode the frame, we will need to create a NV_ENC_PIC_PARAMS structure and set some encoding parameters due to the nature of our design in 5.5.6. Since we are able to encode multiple video sequences without the need to terminate and create a new encoder, we have to signal NVENC whenever the first frame from each video sequence is going to be encoded. The flag we have to send is NV_ENC_PIC_FLAG_OUTPUT_SPSPPS, which forces the encoder to write the sequence and picture header in the encoded bitstream of the frame. Without a header, decoders would not recognize the format thus playback would be impossible. Other parameters which are necessary for encoding are a reference to the input buffer and output buffer. Then we can send the structure to the encoder with nvEncEncodePicture() function provided by NVENC API to start the encoding process. The pointer to the input buffer will be used by the encoder to know where to fetch frame data, then the encoding process will begin when the transfer from system memory to video memory is completed. When a frame has been encoded, it will be stored in the output buffer. Reading from the output buffer requires us lock, thus transferring the encoded bitstream from video memory to system memory. Then we can either write the data to file or store it in memory. Upon completion of processing the output we will have to unlock the output buffer. During encoding a frame, sometimes the encoder needs more data from the user to fully encode a frame, it will then signal us and wait for new data. This behaviour is strictly caused by B-frames, since they need a reference from both a previous frame and a future frame. Like x264, when encoding of every frame has been completed, we will need to flush the encoder to get the remaining encoded frames. The transferring of data between system memory and video memory starts whenever locking is executed as shown in figure 5.8. According to ffprobe, FFmpegs tool for getting metadata from a video file, the output of NVENC is in pixel format YUV 4:2:0.

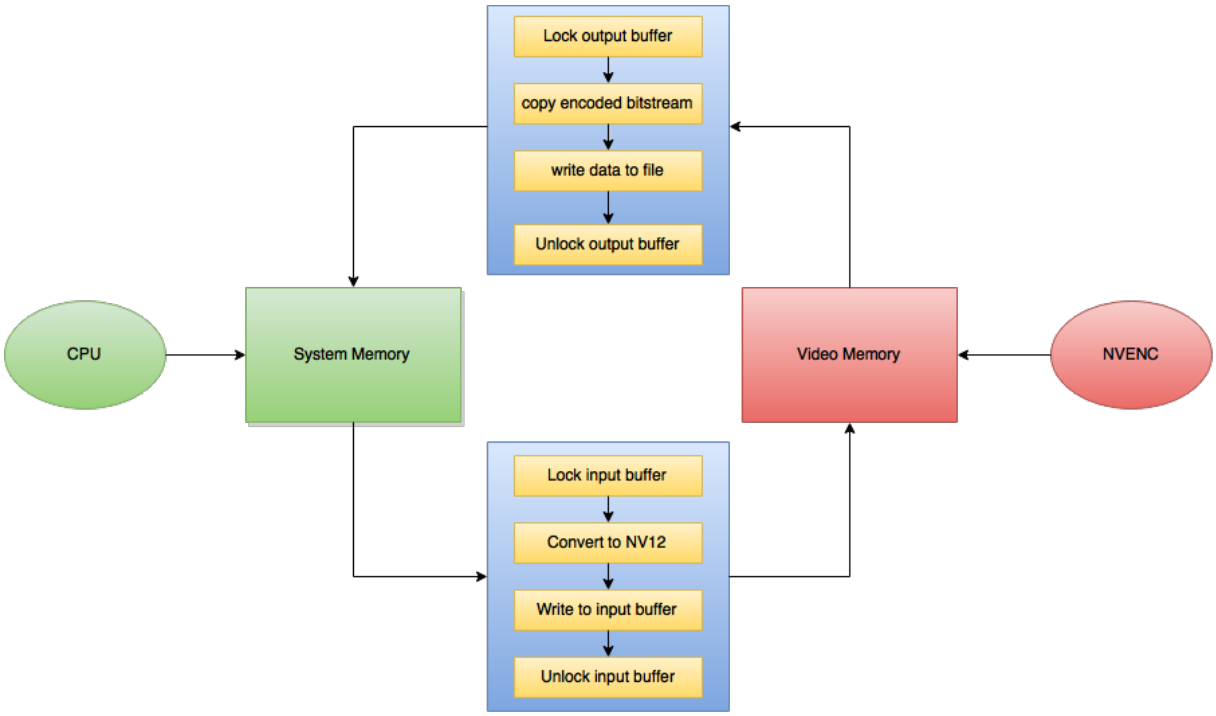



Figure 5.8: Overview of the interaction between CPU and GPU

## 5.7 Summary

In this chapter, we have presented NVIDIAs hardware encoder NVENC. The limitation it had was extensively discussed and the necessity of downgrading drivers to initialize beyond 2 encoding session was mentioned. The tool which supports NVENC can also be used for testing though it has its limitation. During this chapter we have made many optimization and reduced unnecessary data to the program. We examined many the different presets and encoder parameters NVENC supports and how to use them. There were some debate on whether there would be a performance bottleneck between synchronous and asynchronous mode, whatever the case we would still be limited to synchronous mode since the alternative approach is not supported in linux. We demonstrated how to further optimize the encoder for usage in a parallel enviroment by reconfiguring it, instead of terminating and initiating new resources when needed. There were some problems with the pixel formats thus we introduced new image formats and discussed several problems which we encountered. Consequently, we needed to convert the pixel format to the one NVENC supports, some approaches were made and their pros and cons were mentioned. We discovered a limitation to the hardware encoder which did not affect x264. The procedure of encoding a frame was presented in detail.

# Chapter 6

# Case Study: Hardware Encoding

We wanted to be able to cross the threshold of 3 seconds which indicates real-time encoding, but with the usage of the software encoder x264 we could not achieve the it. Thus we wanted to explore new areas to further improve the encoding latency, hence hardware encoders were the most logical route for us. Many were considered such as Intel Quick sync, AMD VCE and NVENC. The hardware encoder from AMD was not chosen because it could not be easily obtained. So we considered the other two, but we decided that our system should be scalable without difficulty thus NVENC was chosen over Quick sync. The reason was we could not have beyond 1 Intel CPU with Quick Sync on-board without specialized equipment. In this chapter we will experiments encoding of the various tiling approaches with NVENC and find out how efficient they are.

## 6.1 NVENC Encoding

The performance measurement still uses the same set-up as before, an Intel Core i7-4700, 8GB and SSD hardisk. However, there is also the additional hardware encoder NVENC that we used, which is integrated on a GeForce GTX750 Ti. We used NVENC SDK 3.0 with NVIDIA driver version 334.21 and Cuda compilation tools, release 6.0, V6.0.1. .

### 6.1.1 Limitation

NVIDIA has stopped supporting GeForce and low-end Quadro GPUs, thus the use of nvidia-smi(used for profiling) are showing only limited informations. Thus we could not get a measurement directly from NVENC, which gives us the only way to assess the hardware encoder is through encoding latency. Memory usage could also be measured, but after some extensive testing, there were no correlation between the encoding efficiency and RAM used. Thus we would only mentioned and shows some figures at the beginning, but they will be excluded in the later experiments.

### 6.1.2 Experiment 9: Encoding a Panorama Video into multiple qualities with NVENC

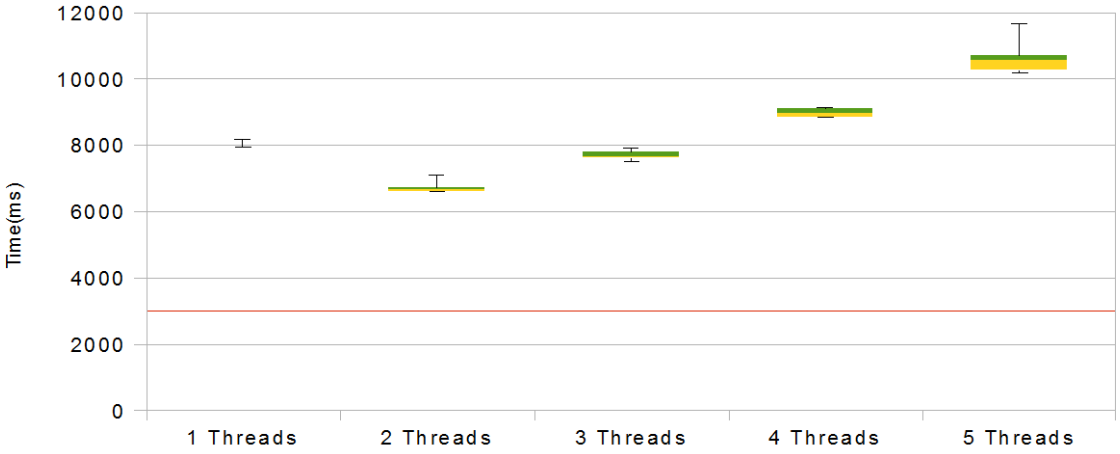



Figure 6.1: Encoding panorama video using NVENC

The restriction we had during testing was the limit of 5 threads. It was due to the fact there was only 5 video streams that could be encoded at the same time. Thus we experimented with 1-5 threads as seen in figure 6.1. From the same figure we can spot a curve from the 1 thread to the 5 threads in , where using 2 threads was the best case. There seems to be an overhead caused by context switching in the GPU by the look of the increased latency in accordance to threads used.

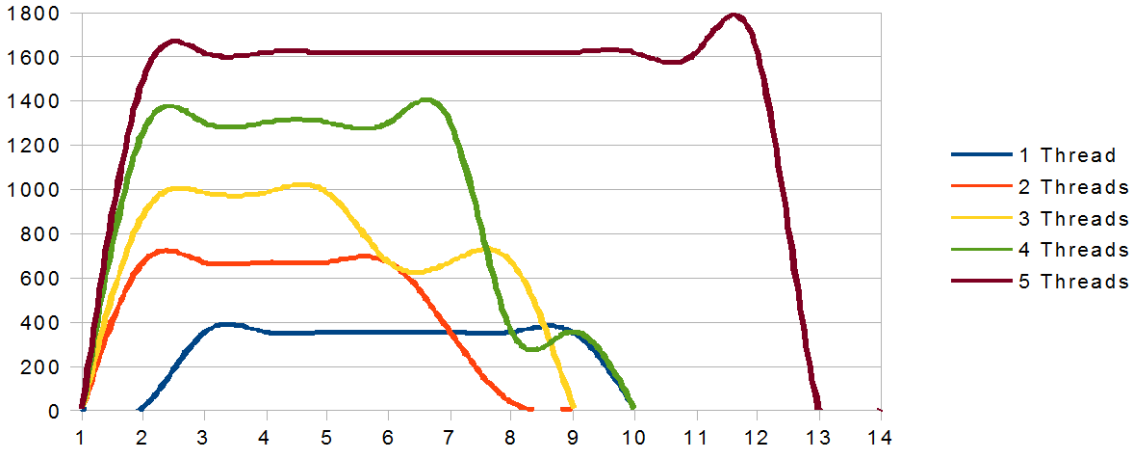



Figure 6.2: Memory usage

We can observe there is a high usage of memory in figure 6.2 following the number of threads dedicated for encoding, but the latency increased the more memory we used. The reason is unknown at the moment.

### 6.1.3 Experiment 10: Encoding 2X2 Tiles into multiple qualities with NVENC

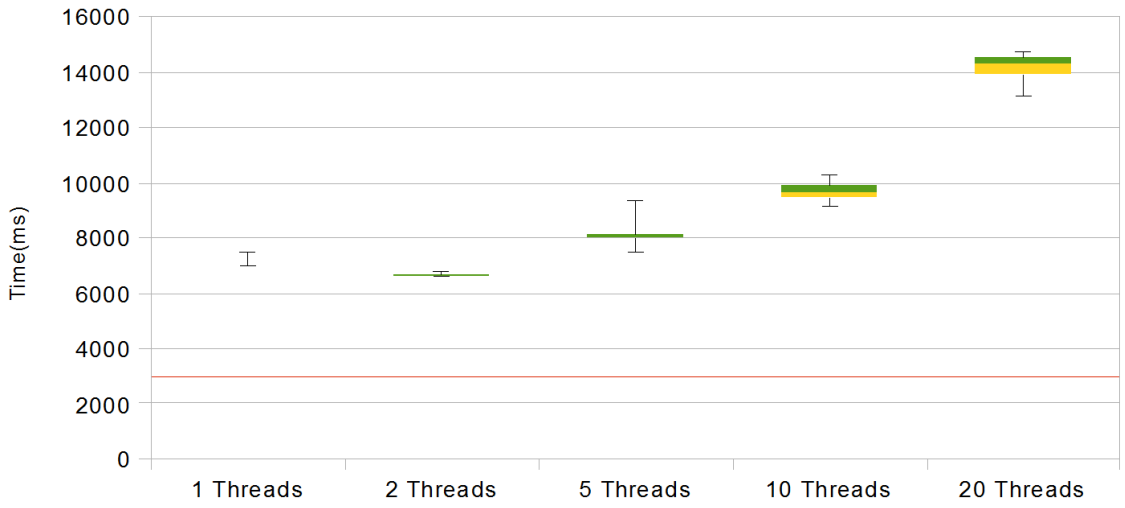



Figure 6.3: Encoding 2X2 tiles using NVENC

There is a slight difference between the 1 thread and 3 threads compared to the previous experiment as seen in figure 6.3. There is also the test with 20 threads which further proved our suspicion of CUDA context switching creating overheads. We expected the performance should be on par or close to the x264 encoder, but the results compared to x264 indicated differently. We could only test 20 threads due to insufficient tasks to initiate more threads.

Table 6.1: CPU, GPU and memory usage

| GPU Memory | 5,62% | 9,38% | 20,81% | 39,47% | 77,09% |
|---|---|---|---|---|---|
| CPU | 91,88% | 101,11% | 109,8% | 141,27% | 139,56% |
| RAM | 53,74% | 51,37% | 49,33% | 55,5% | 71,19 |

The RAM usage increases in parallel with the number of threads, but the usage of GPU memory does not present any performance. Unlike the the software encoder we implemented, the usage of system memory is higher than every experiment that was made previously(in x264). The difference can be observed in table 6.1. The reason for high usage of memory is because a frame need to be in the GPU memory before NVENC can proceed to process it, thus we have to transfer the frame data from system memory to GPU memory.

### 6.1.4 Experiment 11: Encoding 4X4 Tiles into multiple qualities with NVENC

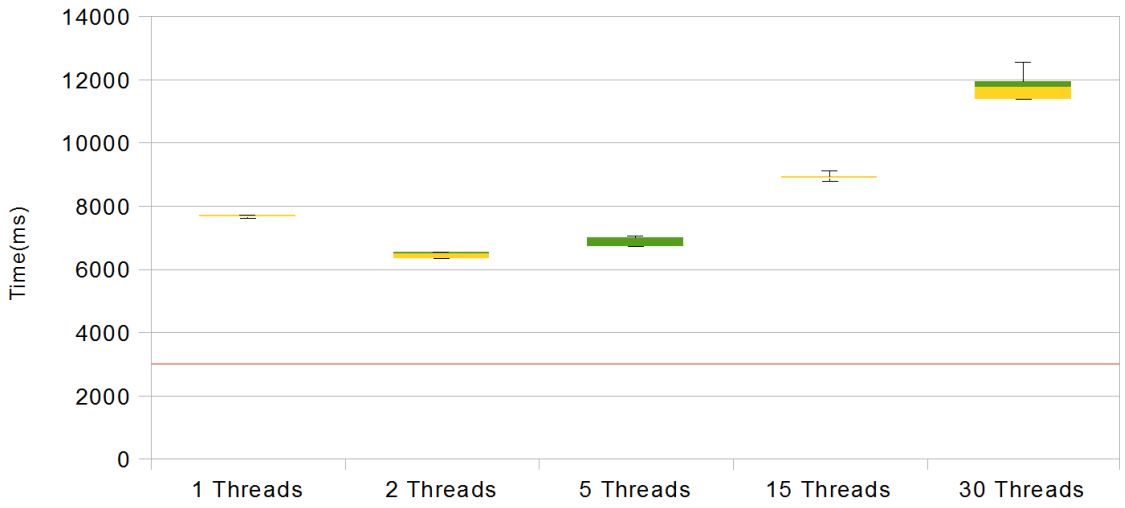



Figure 6.4: Encoding 4X4 tiles using NVENC

We wanted to find the threshold where the GPU would break and crash, and adapt the number of threads until we found a number where the program is stable. The highest number we could achieve was 30 threads, dedicating more threads beyond 30 would crash immediately. We can see the same curve in fiugre 6.4 as in all previous experiments with NVENC. Using 1 thread or beyond 2 threads had results that deviated a little, but there was not any noticeable change with 2 threads. We could actually claim that it was non-existent.

### 6.1.5 Experiment 12: Encoding 8X8 Tiles into multiple qualities with NVENC

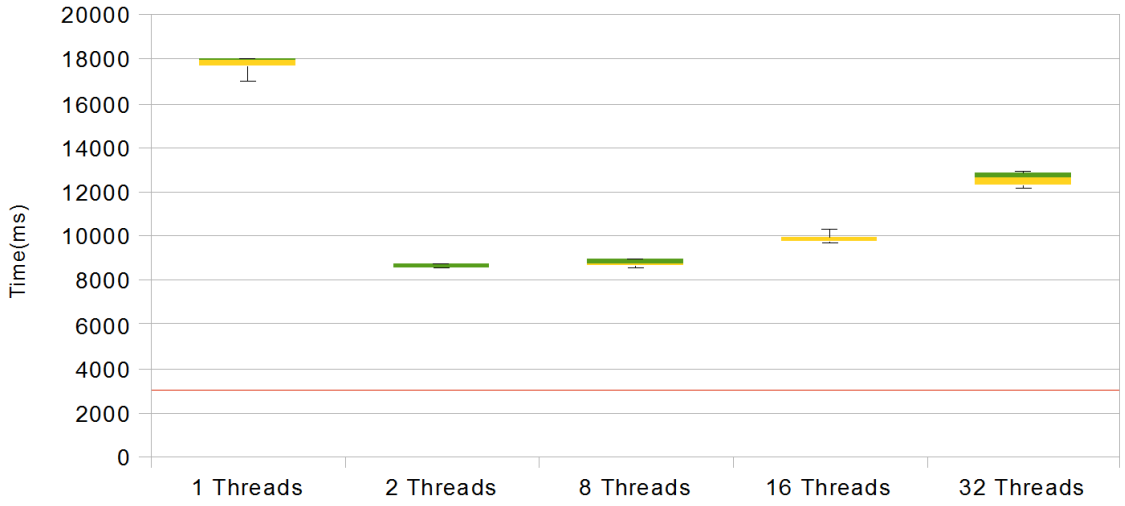



Figure 6.5: Encoding 8X8 tiles using NVENC

According to figure 6.5, NVENC is not performing well with small tasks which can be observed by the increased encoding latency. According to the portable document format(PDF) that followed the NVENC SDK 3.0 and 5.0.1 packages[38]. It is documented that NVENC supports

multiple hardware encoding but due to hardware context switching penalties, there is a slight drop in the encoding performance, and the penalty will increase in parallel with then number of concurrent sessions. But there seems to be variants of the new Maxwell architecture that has an additional instance of NVENC. If that is the case, then the curve that can observed in all experiments until, could indicate that NVIDIA GeForce GTX750 Ti has 2 instances of NVENC, thus the best cases from all the test were 2 threads.

## 6.2 Summary

The performance using the hardware encoder did not yield the results we wanted. But there are many factors for the difference in latency. First of all, NVENC only supports NV12 thus it has to convert from YUV 4:2:2 to its native format. We did some profiling and found out that the conversion decreased the encoding time, which was also mentioned in 5.6.1. Secondly, even though transferring from system memory to GPU memory seems to be executed at a very fast rate, there are still overhead produced by it. Another factor was also mentioned in 5.5.4, which is about execution modes that NVENC supports. In synchronous mode, which is the only alternative for Linux users, frames are encoded one at a time and a frame cannot be encoded before the previous frame has been encoded. This can be observed in 6.2 where the memory usage is always constant during encoding. There were some improvements which could be made, such as utilizing CUDA arrays but unfortunately it is still not yet supported in the latest NVENC SDK 5.0.1. NVENC was probably not designed to handle tiling, or such small tasks which can be observed by the increased latency with 8X8.

# Chapter 7

# Case Study: Software Encoding combined with Hardware encoding

## 7.1 Introduction

Experimentation with the hardware encoder NVENC did not yield the results we wanted due overheads such as conversion of pixel format, hardware context switching and restrictions(synchronous mode). But the performance using 2 threads gave us a reason to believe that we can improve our design and implementation further by utilizing every resources available on the current test machine. Thus we will be testing and evaluate the result from encoding with both the hardware and software encoders combined.

## 7.2 Setup

The performance measurement still uses the same set-up as before, an Intel Core i7-4700, 8GB,SSD hardisk and NVENC. The driver versions are still NVENC SDK 3.0 with NVIDIA driver version 334.21 and CUDA 6.0.

## 7.3 Testing the scalability of our design and implementation

In every tests we are using threads + 2 GPU threads even if it is not stated, we are using 2 because it was the one instance which had the best case in every tiling approaches. To get a complete evaluation we added an extra step for conversion in the encoding pipeline in x264. Thus we will get only video stream with pixel format YUV 4:2:0.

### 7.3.1 Experiment 13: Encoding panorama video with x264 and NVENC combined

Table 7.1: Experiment 13: CPU and memory measurement for x264 combined with NVENC

| 1 Thread | | 2 Threads | | 3 Threads | |
|---|---|---|---|---|---|
| CPU | RAM | CPU | RAM | CPU | RAM |
| 209,25% | 51,8% | 324,67% | 51,37% | 472,33% | 55,83% |

We can observe a significant increase in encoding time by using both CPU and GPU for processing videos in figure 7.1. The 3 threads + 2 GPU threads were the best case when encoding only panorama video without tiling. The reason for the high utilization of CPU is we had 2 threads, each dedicated to 1 GPU thread. Thus we are actually 3, 4 and 5 threads in the tests. Memory usage is due to transporting data between system and video memory.

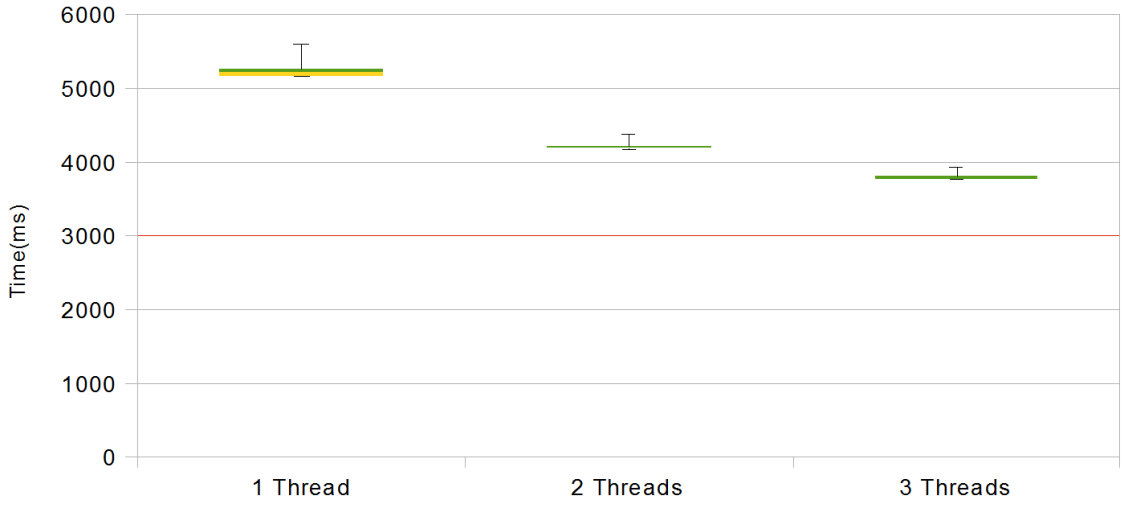



Figure 7.1: Encoding Panorama video with x264 and NVENC

### 7.3.2 Experiment 14: Encoding 2X2 tiles with x264 and NVENC combined

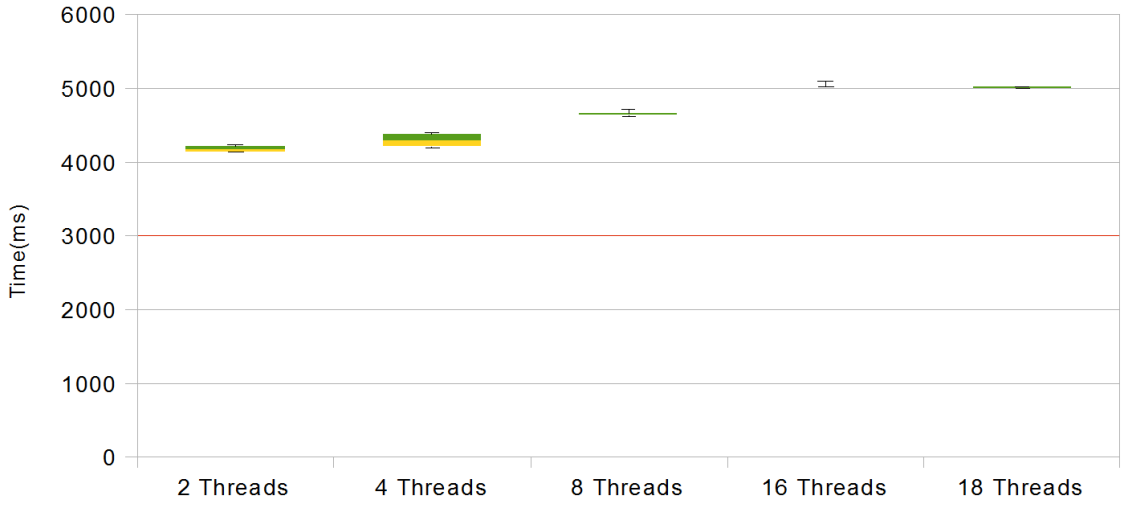



Figure 7.2: Encoding 2X2 tiles with x264 and NVENC

Table 7.2: Experiment 14: CPU and memory measurement for x264 combined with NVENC

| 2 Threads | | 4 Threads | | 8 Threads | | 16 Threads | | 18 Threads | |
|---|---|---|---|---|---|---|---|---|---|
| CPU | RAM | CPU | RAM | CPU | RAM | CPU | RAM | CPU | RAM |
| 338,33% | 47,97% | 519,5% | 40,38% | 784,25% | 34,05% | 788,25% | 33,93% | 797,33% | 35,37% |

The encoding latency increases in parallel with the number of threads can be observed in figure 7.2. Looking at the table 7.2, we can can see the memory usage was reduced significantly when the number of threads increased. As we know, the high memory usage is due to the fact that we have to transfer frame data to GPU. Thus low memory is equivalent to no data transferred, meaning NVENC is idle. Another factor which can confirm our deduction is the fact we have a low amount of tasks available in 2X2 tiling approach(20 tasks total, 4 tiles*5

qualities). Thus increasing the number of threads will reduce the tasks at a much higher rate. With 18 threads + 2 GPU threads, all tasks will be taken straight away. With 2 threads as the best case and 18 threads as the worst case. Additionally the fact that encoding latency increases by the number of threads and the results from the previous experiment. We can conclude that NVENC is much faster at processing data with a huge surface area.

### 7.3.3 Experiment 15: Encoding 4X4 tiles with x264 and NVENC combined

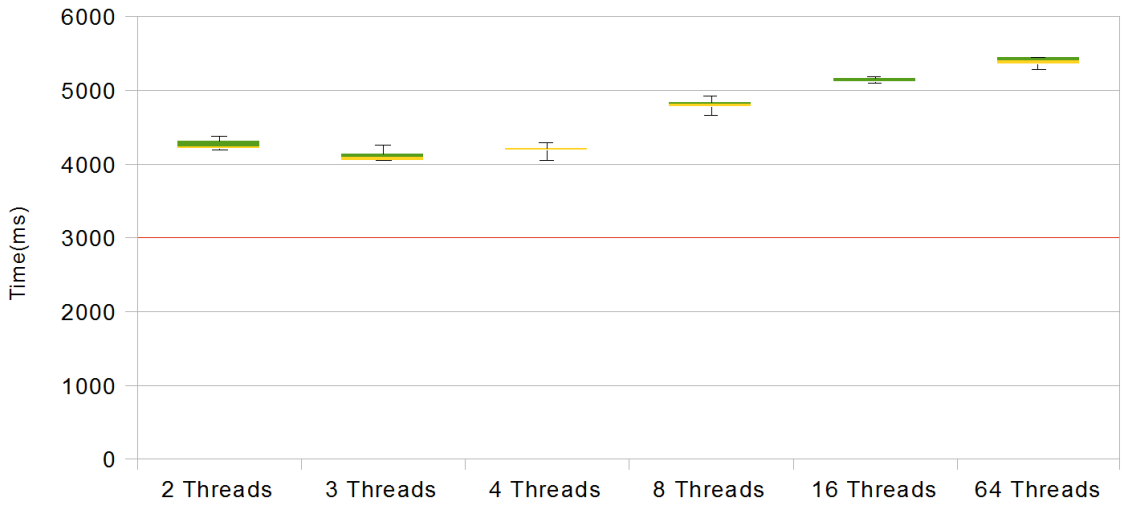



Figure 7.3: Encoding 4X4 tiles with x264 and NVENC

Table 7.3: Experiment 15: CPU and memory measurement for x264 combined with NVENC

| Threads | 2 | 3 | 4 | 8 | 16 | 64 |
|---|---|---|---|---|---|---|
| CPU | 326,25% | 478,67% | 582,33% | 772,25% | 785% | 763% |
| RAM | 43,5% | 37,47% | 39,43% | 28,28% | 23,63% | 23,83 |

We can confirm our deduction that NVENC is more effective in processing large resolution images by observing figure 7.3. The 2 threads test which was best case in 2X2 tiling performed mediocre in 4X4 approach, and by adding an additional thread there was an increase and best case performance for this test. Experimenting with 4 threads and beyond the encoding latency increases in parallel with number of dedicated threads.

### 7.3.4 Experiment 16: Encoding 8X8 tiles with x264 and NVENC combined

Table 7.4: Experiment 17: Encoding 8X8 tiles with x264 and NVENC combined

| Threads | 2 | 3 | 4 | 8 | 16 | 64 |
|---|---|---|---|---|---|---|
| CPU | 219% | 309,5% | 435,75% | 548% | 738% | 757,5% |
| RAM | 47,8% | 41,08% | 36,33% | 35,63% | 29,2% | 23,08 |

Looking at the graph in figure 7.4, a convex curve can be observed starting from 2 threads to 64 threads, with 4 threads as best case. A slight increase in encoding time can be noticed on every thread instances, due to the vast amount of small tasks. However, because IO operations and waiting time for getting mutexes has been reduced significantly as a consequence of smaller tiles.

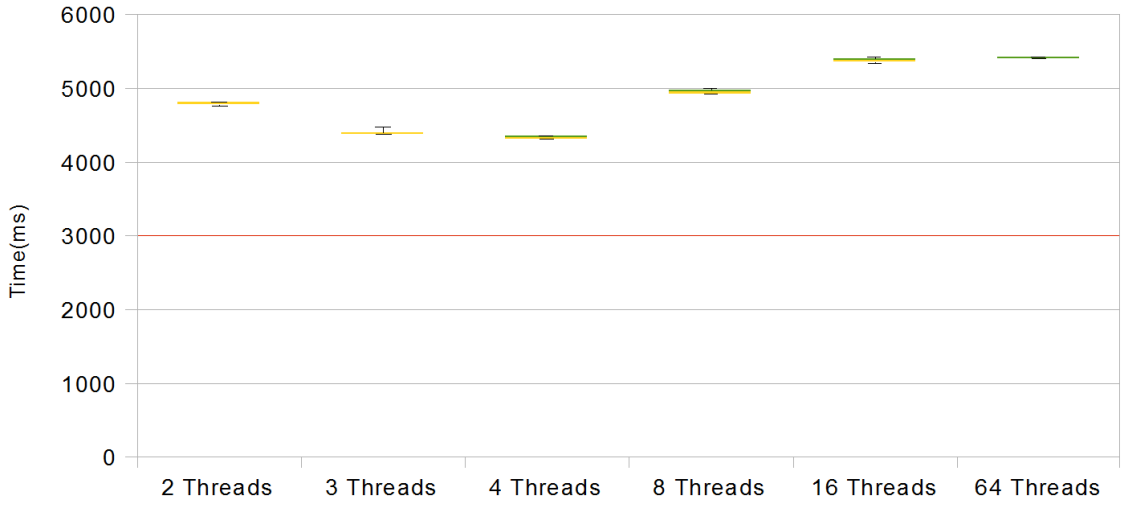



Figure 7.4: Encoding Panorama video with x264 and NVENC

## 7.4 Summary

In this chapter, we have presented experiments which confirms to us that the threshold for real-time encoding is achievable. We are only using one GPU with NVENC, and it already improved the encoding time significantly. Thus we have shown that our solution are scalable with the use of additional hardware in encoding. Though not proven scientifically, there are indications which makes us believe that NVENC is better at processing high resolution images.

# Chapter 8

# Conlusion

In this chapter, we will look back and summarize our work, mention our main contributions and lastly we will look into ways for future improvements of our work.

## 8.1 Summary

In chapter 1, we defined the problem we had which we tried to solve by design and implementing our system. The challenges we had to face was finding the overhead caused by tiling, the storage requirement and whether it is possible to encode it in real time. We tried to solve it by using the free video processing tool FFmpeg, and using libx264 which is a wrapper to the H.264 encoder x264.

### 8.1.1 Software Encoder: Design and Implementation

In chapter 3, we discussed several techniques for reducing the storage consumption thus bandwidth, and tested several encoding parameters to find out the most suited configuration for our design and implementation. We came over some issues when trying to use the free video processing tool FFmpeg. Migrating from sequential encoding over to parallel encoding revealed that the libraries from FFmpeg had some global variables that was shared between functions, which resulted in corrupted data. But the issue was solved by implementing a lock which could be used by FFmpeg for synchronization. We have also presented a detailed description to replicate the prototype used for testing.

### 8.1.2 Case Study 4: Software Encoding with libx264

In chapter 4, we experimented with various encodings schemes with various tiling approaches. We discovered with sequential encoding that system resources was too inefficiently used, thus we tried to apply frame slicing. It was an optimization in FFmpeg where a frame was separated into smaller frames, and create and dedicate a child thread for each sub-frame. It yielded much better performance, but it was discovered that it could not satisfy our requirement of tiling. The reason is when increasing the number of tiles, the performance decreased significantly. We found the overhead was due to the termination and creation of threads, inside a software encoder. Frame slicing uses persistent threads which are on standby if there are no tasks available, thus it should not produce considerable overhead. The encoders in FFmpeg will terminate when a stream has been encoded, thus threads which was created inside the encoder will also be terminated. So we concluded that sequential encoding with or without frame slicing would not suffice for our design.

Experimenting with a given number of threads based on the tiling approach gave us an indication of the overhead produced during tiling. We learned that each tiling approach has its own number of threads which would yield the best case. For 2X2 and 4X4 tile formation, using 8 threads would have the best performance, but in 8x8 tiles 4 threads would have the

best case. Even though video encoding is CPU-bounded, where the same amount of threads as CPU cores will have least encoding time, there are many other factors that we discovered which affects the result. With using 2X2 of 4X4 tiles, an encoder has to use more time to process the whole surface of an image. Thus during this time, other threads will have to wait. This waiting mechanism was actually initiated by us to prevent data corruption, but the waiting time for a mutex made threads idle, thus the increase in encoding latency. IO operations was another factor which also applies here. Usually a thread would context switch to release system resources for other threads to fetch, but there are no threads waiting for system resources when applying 4 threads for encoding of 2X2 and 4X4 tiles. However, using 8X8 tiles 4 threads was the best case, since the waiting time for mutexes and IO operations has been drastically reduced as a consequence of lower processing time to encode each tile. Though the results we got from the different test did satisfy the 3 second threshold indicating real time encoding, we did prove that adaptive encoding of tiles reduced the bandwidth requirement by a huge margin, by exploiting the Region of Interest in virtual cameras of course. We also provided another settings which would further reduce the storage consumption, but the trade off is between encoding latency and compression efficiency.

### 8.1.3 Hardware Encoding: Design and Implementation

In chapter 5, we proposed the usage of hardware encoders to find if there are any difference between specialized hardware and general purpose CPU which used the software encoder x264. We encountered a slight problem during our implementation of the hardware encoder due to a restriction that was applied to the GeForce and low Quadro GPUs with the new NVIDIA drivers. It was solved but it was a necessity to acquire a licence key and downgrade several drivers for NVENC to work. We have discussed some of the issues with NVENC, such as CUDA context, resolutions bugs, and several tests of different encoders parameters. A detailed overview of how to implement a working prototype was also mentioned.

### 8.1.4 Case Study: Hardware Encoding

In chapter 6, we extensively tested the hardware encoders. There was some issues that was not mentioned in the chapter. It seems if we try to execute the hardware encoder right after each other there could be a bug were allocating a CUDA context would require significant time. The results from this case study were not in our expectation. We thought the performance would be on par or a little lower than x264. But there were many factors which made it like that. Using NVENC we had to convert YUV 4:2:2 to its native pixel format NV12, this conversion would prove to be a huge overhead. We discovered that a CUDA context can be shared among a given number of GPU threads, and how many threads which can run concurrently with NVENC is decided by the size of a tile. In every test using only 2 GPU threads gave us the best performance.

### 8.1.5 Case Study: Software Encoding combined with Hardware Encoding

In chapter 7, we wanted to further increase the performance with the use of both encoders simultaneously. With the experimentation the results we got was a significant decrease in encoding latency. Which can be proven that our system can be scaled with the use of additional hardware.

## 8.2 Main Contributions

During this thesis we have made several contributions related to tiling, cost of overhead and reduction of bandwidth which will be described under.

#### Scalability of the system

We have shown in chapter 7 that our prototype can scale up by integrating hardware component and software component and the performance was significantly better that all previous attempts of optimizations.

#### Tiling module

Our main contribution is the design and implementation of the tiling module, and analysis of the different tiling approaches as described in chapter 3. This module is able to separate a panorama video into a given number of tiles.

#### Analysis of the overhead produced as a consequence of tiling

We had experimented with difference tiling approaches combined with different encoding schemes and analysed the overhead produced during these tests.

#### Techniques for reducing the bandwidth consumption

In chapter 4 we have shown a way to further reduce the consumption of storage thus bandwidth. But the trade off is the increased in encoding latency.

#### Analysis of between hardware encoder and software encoder

We experimented with NVIDIAs hardware encoder NVENC and software encoder x264. We compared the results between them in terms of latency and CPU usage.

#### Paper in proceedings

We have a paper in proceedings for the 21st International Packet Video Workshop (PV 2015)[17]. The paper discusses the overall process of the system, and this thesis emphasises the server side of the system. The discussion about the server side design and implementation in the paper was the thesis initial phase thus it did not have any comparisons with NVENC.

## 8.3 Conclusion

Even though we could not achieve real time encoding with the hardware we used for testing. However, the processing requirement has an upper bound and the number of users does does affect us. Thus we do not have scaling issue. We have proven that the system can be improved significantly by utilizing additional hardware. We have succeeded in showing the overhead produced as a consequence of tiling. We have also presented how much reduction of storage space thus bandwidth can be obtained by using an adaptive tiling approach. Thus our conclusion is that adaptive tiling of panorama video stream can be applied in real life, we have succeeded to satisfy the problems/statements that we declared in chapter 1.

## 8.4 Future Works

#### test the system with larger data sizes

During this thesis, the data sizes were limited due to video segments having only a duration of 3 seconds.

**Using several CPU simultanously**

Video encoding is CPU bounded in most cases as we can observe in this thesis. Thus we could try to do video encoding on xenon processor on a mainboard which supports several CPUs on it.

**Try to combine different specialized hardware**

Computers nowadays has a an integrated GPU on their CPU which can be used for encoding. Thus you could combine a dedicated GPU with the integrated GPU, and if there are still CPU resources left, use the remaining for encoding too. T This property refers to how well a system can scale up by integrating hardware and software components supplied from different designers or vendors. This calls for using components with a standard, open architecture and interface. In the software area, this is called portability.

**HEVC**

High Efficiency Video Coding, which is a successor to H.264/MPEG-4 AVC is said to double the data compression ratio compared to its predecessor. We could apply the file format as a standard in our system to further reduce the production of overhead as a consequence of tiling, thus reducing the bandwidth consumption. x265 encoder is implemented and supported in both ffmpeg and NVENC 5.0.

**NVENC SDK**

Implement it with a newer version of NVENC SDK on a newer hardware, in asynchronous mode. For further optimization experiment with cudaarray, though it is not yet supported.

**Newer Tiling approaches**

A tile does not necessary need to be in square or rectangle form, experiments with different shapes to find the overhead cost and find out the pros and cons.

**Newer Hardware**

The main flaw of hardware encoders are its inability to update it codec, thus if there are new findings where the corresponding codec can be optimized, it cannot be applied to the hardware encoder. Therefore, test the program with newer hardware.

**Different hardware encoders**

We have mentioned some hardware encoders in this thesis. We could try to design and implement a new encoder where we can divide the workload by using more than just one hardware encoder for encoding. This approach seems to have potential to reduce the latency.

# Appendices

# Appendix A

| ffmpeg version 2.5.git Copyright | (c) 2000-2015 the FFmpeg developers |
|---|---|
| built on Feb 12 2015 16:15:06 | with gcc 4.7 (Ubuntu/Linaro 4.7.3-2ubuntu1 12.04 |
| configuration: –enable-libx264 | –enable-nvenc –enable-gpl –enable-nonfree |
| libavutil | 54. 16.100 / 54. 16.100 |
| libavcodec | 56. 20.100 / 56. 20.100 |
| libavformat | 56. 18.101 / 56. 18.101 |
| libavdevice | 56. 4.100 / 56. 4.100 |
| libavfilter | 5. 7.101/ 5. 7.101 |
| libswscale | 3. 1.101 / 3.1.101 |
| libswresample | 1. 1.100 / 1.1.100 |
| libpostproc | 53. 3.100 / 53.,3.100 |

Table 1: FFmpeg version we used and its configuration

Table 2: Machine configurations

| CPU | Intel Core i7-4700 |
|---|---|
| RAM | 8GB |
| DISK | SSD |
| GPU | GeForce GTX750 Ti |
| NVENC API Version | NVENC SDK 3.0 |
| GPU driver version | NVIDIA driver version 334.21 |
| CUDA driver version | CUDA compilation tools, release 6.0, V6.0.1 |

# Appendix B

The implementation can be fetched from the following repository:
https://bitbucket.org/hoangbn/master-implementation

# Bibliography


[1] *4K as standard in 2017*. URL: http://www.trustedreviews.com/news/toshiba-suggests-4k-tvs-will-be-mainstream-by-2017.

[2] *A. Shafiei, Q. M. K. Ngo, R. Guntur, M. K. Saini, C. Pang, and W. T. Ooi, "Jiku live," in Proc. of ACM MM, 2012, p. 1265.*

[3] *Adaptive Bitrate Streaming*. URL: http://en.wikipedia.org/wiki/Adaptive_bitrate_streaming.

[4] *Apple's HLS*. URL: https://developer.apple.com/library/ios/documentation/NetworkingInternet/Conceptual/StreamingMediaGuide/Introduction/Introduction.html.

[5] *Chrominance*. URL: http://en.wikipedia.org/wiki/Chrominance.

[6] *CUDA Context*. URL: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#context.

[7] *Detailed Overview Of NVENC ENcoder API*. URL: http://on-demand.gputechconf.com/gtc/2014/presentations/S4654-detailed-overview-nvenc-encoder-api.pdf.

[8] *Difference between Sequential and Parallel Programming*. URL: https://mivuletech.wordpress.com/2011/01/12/difference-between-sequential-and-parallel-programming/.

[9] *Different YUV*. URL: http://www.fourcc.org/yuv.php.

[10] *Douglas E Comer, David Gries, Michael C Mulder, Allen Tucker, A Joe Turner, Paul R Young, and Peter J Denning. "Computing as a discipline". In: Communications of the ACM 32.1 (1989), pp. 9–23.*

[11] *FFmpeg Locking Mechanism*. URL: http://www.mail-archive.com/libav-user@mplayerhq.hu/msg03730.html.

[12] *FFmpeg Locking Mechanism solution*. URL: http://stackoverflow.com/questions/15366441/ffmpeg-which-functions-are-multithreading-safe.

[13] *Ffmpeg thread safety*. URL: http://libav-users.943685.n4.nabble.com/Threads-safe-issue-with-ffmpeg-c-td945683.html.

[14] *FFmpeg threading optimimzation implementation - Frame slicing*. URL: http://ffmpeg.org/doxygen/trunk/frame__thread__encoder_8c_source.html#l00227.

[15] *FFmpeg with NVENC support*. URL: https://github.com/Brainiarc7/ffmpeg_libnvenc.

[16] *File writing*. URL: http://www.cplusplus.com/reference/fstream/fstream/.

[17] Vamsidhar Reddy Gaddam et al. 'Tiling of Panorama Video for Interactive Virtual Cameras: Overheads and Potential Bandwidth Requirement Reduction'. In: (2014).

[18] *H.264*. URL: https://en.wikipedia.org/wiki/H.264/MPEG-4_AVC.

[19] *H.264 Encoders*. URL: http://compression.ru/video/codec_comparison/h264_2012/mpeg4_avc_h264_video_codecs_comparison.pdf.

[20] Pål Halvorsen et al. *Efficient Implementation and Processing of a Real-time Panorama Video Pipeline*. 2013. URL: http://home.ifi.uio.no/paalh/publications/files/mmsys2013-bagadus.pdf.

[21] Brian Harvey and Matthew Wright. *Sequential Programming*. 1999. URL: https://www.eecs.berkeley.edu/~bh/ssch20/part6.html.

[22] *HTTP Live Streaming*. URL: http://en.wikipedia.org/wiki/HTTP_Live_Streaming.

[23] Wei Tsang Ooi Hui Wang Vu-Thanh Nguyen and Mun Choon Chan. 'Mixing Tile Resolutions in Tiled Video: A Perceptual Quality Assessment'. In: (2014).

[24] *Hyper-Threading*. URL: https://en.wikipedia.org/wiki/Hyper-threading.

[25] *IBP Reordering*. URL: http://blog.monogram.sk/janos/2008/06/08/b-frames-in-directshow/.

[26] *Images and Pixels*. URL: https://processing.org/tutorials/pixels/.

[27] *Images and Pixels*. URL: https://processing.org/tutorials/pixels/.

[28] *ISO/IEC JTC 1*. URL: https://en.wikipedia.org/wiki/ISO/IEC_JTC_1.

[29] *ITU Telecommunication Standardization Sector Video Coding Experts Group*. URL: https://en.wikipedia.org/wiki/ITU-T.

[30] *K. Q. M. Ngo, R. Guntur, and W. T. Ooi, "Adaptive encoding of zoomable video streams based on user access pattern," in Proc. of MMSys, 2011, p. 211.*

[31] Hassan Bin Tariq Khan and Malik Khurram Anwar. *Quality-aware frame skipping for MPEG-2 video based on interframe similarity*. URL: http://www.idt.mdh.se/utbildning/exjobb/files/TR0527.pdf.

[32] *M. A. Wilhelmsen, H. K. Stensland, V. R. Gaddam, P. Halvorsen, and C. Griwodz, "Performance and Application of the NVIDIA NVENC H.264 Encoder,"* URL: http://on-demand.gputechconf.com/gtc/2014/poster/pdf/P4188_real-time_panorama_video_NVENC.pdf.

[33] Microsoft. *Image Stride*. URL: https://msdn.microsoft.com/en-us/library/windows/desktop/aa473780(v=vs.85).aspx.

[34] Microsoft. *YUV 8-Bit YUV formats for Video Rendering*. 2008. URL: https://msdn.microsoft.com/en-us/library/windows/desktop/dd206750(v=vs.85).aspx.

[35] *Mutual exclusion*. URL: https://en.wikipedia.org/wiki/Mutual_exclusion.

[36] *NVENC limitation*. URL: https://devtalk.nvidia.com/default/topic/800942/gpu-accelerated-libraries/session-count-limitation-for-nvenc-no-maxwell-gpus-with-2-nevenc-sessions-/.

[37] NVIDIA. *NVENC - NVIDIA hardware video encoder*. https://developer.nvidia.com/nvidia-video-codec-sdk.

[38] NVIDIA. *NVENC - NVIDIA hardware video encoder*. http://developer.download.nvidia.com/compute/nvenc/v5.0_beta/NVENC_DA-06209-001_v06.pdf. 2014.

[39] NVIDIA. *NVENC - NVIDIA Performance*. https://developer.nvidia.com/sites/default/files/akamai/cuda/files/CUDADownloads/NVENC_AppNote.pdf.

[40] NVIDIA. *NVIDIA GTX750 TI*. http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce-GTX-750-Ti-Whitepaper.pdf. 2014.

[41] *P. Halvorsen, S. Sægrov, A. Mortensen, D. K. Kristensen, A. Eichhorn, M. Stenhaug, S. Dahl, H. K. Stensland, V. R. Gaddam, C. Griwodz, and D. Johansen, "Bagadus: An integrated system for arena sports analytics – a soccer case study," in Proc. of ACM MMSys, Mar. 2013, pp. 48–59.*

[42] *Parallel computing*. URL: http://en.wikipedia.org/wiki/Parallel_computing.

[43] *Pipeline(Computing*. URL: https://en.wikipedia.org/wiki/Pipeline_(computing).

[44] *QPSNR and SSIM program*. URL: http://qpsnr.youlink.org/.

[45] *R. Guntur and W. T. Ooi, "On tile assignment for region-of-interest video streaming in a wireless LAN," in Proc. of NOSSDAV, 2012, p. 59.*

[46] *Race Condition*. URL: https://en.wikipedia.org/wiki/Race_condition.

[47] Werner Robitza. *Constant Rate Factor*. URL: http://slhck.info/articles/crf.

[48] *SavantSmartView*. URL: http://dealers.savantav.com/portal/SavantSandbox/Released%20User%20Guides/009-1108-00%20SmartView%20Tiling%20User%20Guide.pdf.

[49] *Scaling virtual camera services to a large number of users*. URL: http://home.ifi.uio.no/paalh/publications/files/mmsys2015-gaddam.pdf.

[50] Marius Tennøe et al. *Efficient Implementation and Processing of a Real-time Panorama Video Pipeline*. 2013. URL: http://home.ifi.uio.no/paalh/publications/files/ism2013-bagadus.pdf.

[51] *Transcoding*. URL: http://en.wikipedia.org/wiki/Transcoding.

[52] *Understanding YUV formats*. URL: https://www.ptgrey.com/KB/10092.

[53] *V. R. Gaddam, R. Langseth, S. Ljødal, P. Gurdjos, V. Charvillat, C. Griwodz, and P. Halvorsen, "Interactive zoom and panning from live panoramic video," in Proc. of ACM NOSSDAV, 2014, pp. 19:19–19:24.*

[54] *Video Codec*. URL: https://en.wikipedia.org/wiki/Video_codec.

[55] *Video Encoding General*. URL: http://www.heywatchencoding.com/what-is-video-encoding.

[56] *Video Encoding General 2*. URL: http://help.encoding.com/knowledge-base/article/what-is-video-encoding/.

[57] *W.-C. Feng, T. Dang, J. Kassebaum, and T. Bauman. Supporting region-of-interest cropping through constrained compression. ACM Transactions on Multimedia Computing, Communications and Applications, 7(3):17:1–17:16, Aug. 2011.*

[58] *x264 Presets*. URL: https://trac.ffmpeg.org/wiki/Encode/H.264.

[59] *YUV*. URL: http://en.wikipedia.org/wiki/YUV.